

General Track

2004 USENIX Annual Technical Conference

Boston, MA, USA

June 27–July 2, 2004

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: <http://www.usenix.org>

The price is \$35 for members and \$45 for nonmembers.
Outside the U.S.A. and Canada, please add
\$15 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

2003 San Antonio, TX	1990 Summer Anaheim
2002 Monterey, CA	1990 Winter Washington, D.C.
2001 Boston, MA	1989 Summer Baltimore
2000 San Diego, CA	1989 Winter San Diego
1999 Monterey CA	1988 Summer San Francisco
1998 New Orleans	1988 Winter Dallas
1997 Anaheim	1987 Summer Phoenix
1996 San Diego	1987 Winter Washington, D.C.
1995 New Orleans	1986 Summer Atlanta
1994 Summer Boston	1986 Winter Denver
1994 Winter San Francisco	1985 Summer Portland
1993 Summer Cincinnati	1985 Winter Dallas
1993 Winter San Diego	1984 Summer Salt Lake City
1992 Summer San Antonio	1984 Winter Washington, D.C.
1992 Winter San Francisco	1983 Summer Toronto
1991 Summer Nashville	1983 Winter San Diego
1991 Winter Dallas	

© 2004 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-21-8

USENIX Association

**Proceedings of the
General Track**

2004 USENIX Annual Technical Conference

**June 27–July 2, 2004
Boston, MA, USA**

Conference Organizers

Program Chairs

Andrea Arpaci-Dusseau, U. Wisconsin, Madison
Remzi Arpaci-Dusseau, U. Wisconsin, Madison

Program Committee

Atul Adya, Microsoft Research
Fay Chang, Google
Fred Douglass, IBM T.J. Watson
Ed Felten, Princeton University
Armando Fox, Stanford University
Val Henson, Sun Microsystems
Wilson Hsieh, University of Utah
Scott Kaplan, Amherst College
Eddie Kohler, University of California, Los Angeles
Vivek Pai, Princeton University
Srinivas Seshan, Carnegie Mellon University
Carl Staelin, Hewlett-Packard Labs Israel
Werner Vogels, Cornell University
Honesty Young, IBM Almaden
Yuan Yuan Zhou, University of Illinois

Daily Plenary Sessions Organizers

Peter H. Salus, Author and Consultant
Ellie Young, USENIX

“The Guru Is In” Coordinator

Clem Cole, Ammasso

The USENIX Association Staff

External Reviewers

Mukesh Agrawal	Adina Crainiceanu	Sneha Kaser	Erik Riedel
Aditya Akella	Chuck Cranor	Kim Keeton	Sami Rollins
Ben Atkins	Timothy Denehy	Sam King	Stanislav Rost
Godmar Back	David Dewitt	Christos Kozyrakis	Mema Roussopoulos
Lakshmi Bairavasundaram	John Douceur	Sanjeev Kumar	Ant Rowstron
Suman Banerjee	Dawson Engler	Monica Lam	Shai Rubin
Mayank Bawa	Kathleen Fisher	Marc Langheinrich	Dan Simon
Rob von Behren	Matthew Flatt	Zhenmin Li	Muthian Sivathanu
John Bent	Douglas Freimuth	Jacob R. Lorch	Alex Snoeren
Ashwin Bharambe	Vinod Ganapathy	David Lowell	Dave Sullivan
Bill Bolosky	Greg Ganger	Qin Lu	Sai Susarla
Nathan Burnett	Tal Garfinkel	Xiaonan Ma	John Tracey
Mike Burrows	Sanjay Ghemawat	Gurmeet Singh Manku	Patrick Tullmann
George Candea	Jon Giffin	Erik Meijer	Deborah Wallach
John Carter	Pawan Goyal	Kobus van der Merwe	Helen J. Wang
Ronnie Chaiken	Steve Gribble	Nirmal Mukhi	Jun Wang
Jim Challenger	Robert Griesemer	Suman Nath	Limin Wang
Abhishek Chandra	Mark Gritter	James Nugent	Justin Weisz
Ranveer Chandra	Indranil Gupta	Jitendra Padhye	Matt Welsh
Girish P. Chandranmenon	Jon Howell	Jeff Pang	Chris Welty
Michael Chapman-Martin	Andy Huang	Shankar Ponnkanti	Joel L. Wolf
Yatin Chawathe	Lan Huang	Florentina Popovici	Alec Wolman
Peter Chen	Gianluca Iannaccone	Vijayan Prabhakaran	Kun Lung Wu
Robin Chen	Liviu Iftode	David Presotto	Jian Yin
Ying Chen	Sitaram Iyer	Niels Provos	Joe Zachary
Zhifeng Chen	Trent Jaeger	Lili Qiu	Vic Zandy
Mihai Christodorescu	Soranun Jiwassurat	Ananth Rao	
Brent Chun	Glenn Judd	John Regehr	

2004 USENIX Annual Technical Conference
General Track
June 27–July 2, 2004
Boston, MA, USA

Index of Authors vii

Message from the Program Chair ix

Monday, June 28, 2004

Instrumentation and Debugging

Session Chair: Val Henson, Sun Microsystems

Making the “Box” Transparent: System Call Performance as a First-Class Result 1
Yaoping Ruan and Vivek Pai, Princeton University

Dynamic Instrumentation of Production Systems 15
Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, Sun Microsystems

Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging 29
Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou, University of Illinois, Urbana-Champaign

Swimming in a Sea of Data

Session Chair: Yuanyuan Zhou, UIUC

Email Prioritization: Reducing Delays on Legitimate Mail Caused by Junk Mail 45
Dan Twining, Matthew M. Williamson, Miranda J.F. Mowbray, and Maher Rahmouni, Hewlett-Packard Labs

Redundancy Elimination Within Large Collections of Files 59
Purushottam Kulkarni, University of Massachusetts; Fred Douglass, Jason LaVoie, and John M. Tracey, IBM T.J. Watson Research Center

Alternatives for Detecting Redundancy in Storage Systems Data 73
Calicrates Policroniades and Ian Pratt, Cambridge University

Network Performance

Session Chair: Carl Staelin, HP Labs

Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying 87
Yu-Chung Cheng, University of California, San Diego; Urs Hölzle and Neal Cardwell, Google; Stefan Savage and Geoffrey M. Voelker, University of California, San Diego

A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths .. 99
Ming Zhang and Junwen Lai, Princeton University; Arvind Krishnamurthy, Yale University; Larry Peterson and Randolph Wang, Princeton University

Multihoming Performance Benefits: An Experimental Evaluation of Practical Enterprise Strategies 113
Aditya Akella and Srinivasan Seshan, Carnegie Mellon University; Anees Shaikh, IBM T.J. Watson Research Center

Tuesday, June 29, 2004

Overlays in Practice

Session Chair: Fred Douglass, IBM Research

Handling Churn in a DHT 127

Sean Rhea and Dennis Geels, University of California, Berkeley; Timothy Roscoe, Intel Research, Berkeley; John Kubiawicz, University of California, Berkeley

A Network Positioning System for the Internet 141

T.S. Eugene Ng, Rice University; Hui Zhang, Carnegie Mellon University

Early Experience with an Internet Broadcast System Based on Overlay Multicast 155

Yang-hua Chu and Aditya Ganjam, Carnegie Mellon University; T.S. Eugene Ng, Rice University; Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang, Carnegie Mellon University

Secure Services

Session Chair: Atul Adya, Microsoft Research

Reliability and Security in the CoDeeN Content Distribution Network 171

Limin Wang, Kyoungsoo Park, Ruoming Pang, Vivek Pai, and Larry Peterson, Princeton University

Building Secure High-Performance Web Services with OKWS 185

Maxwell Krohn, MIT

REX: Secure, Extensible Remote Execution 199

Michael Kaminsky and Eric Peterson, MIT; Daniel B. Giffin, NYU; Kevin Fu, MIT; David Mazières, NYU; M. Frans Kaashoek, MIT

The Network-Application Interface

Session Chair: Vivek Pai, Princeton University

Network Subsystems Reloaded: A High-Performance, Defensible Network Subsystem 213

Anshuman Sinha, Sandeep Sarat, and Jonathan S. Shapiro, Johns Hopkins University

accept()able Strategies for Improving Web Server Performance 227

Tim Brecht, David Pariag, and Louay Gammo, University of Waterloo

Lazy Asynchronous I/O for Event-Driven Servers 241

Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox, Rice University; Willy Zwaenepoel, EPFL, Lausanne

Wednesday, June 30, 2004

Unplugged

Session Chair: Scott F. Kaplan, Amherst

Energy Efficient Prefetching and Caching 255

Athanasios E. Papathanasiou and Michael L. Scott, University of Rochester

Time-based Fairness Improves Performance in Multi-Rate WLANs 269

Godfrey Tan and John Guttag, MIT

EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks 283

*Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin,
University of California, Los Angeles*

Index of Authors

Akella, Aditya	113	Papathanasiou, Athanasios E.	255
Andrews, Christopher R.	29	Pariag, David	227
Brecht, Tim	227	Park, KyoungSoo	171
Cantrill, Bryan M.	15	Peterson, Eric	199
Cardwell, Neal	87	Peterson, Larry	99, 171
Cerpa, Alberto	283	Policroniades, Calicrates	73
Chanda, Anupam	241	Pratt, Ian	73
Cheng, Yu-Chung	87	Rahmouni, Maher	45
Chu, Yang-hua	155	Ramanathan, Nithya	283
Cox, Alan L.	241	Rao, Sanjay G.	155
Douglis, Fred	59	Rhea, Sean	127
Elmeleegy, Khaled	241	Roscoe, Timothy	127
Elson, Jeremy	283	Ruan, Yaoping	1
Estrin, Deborah	283	Sarat, Sandeep	213
Fu, Kevin	199	Savage, Stefan	87
Gammo, Louay	227	Scott, Michael L.	255
Ganjam, Aditya	155	Seshan, Srinivasan	113
Geels, Dennis	127	Shaikh, Anees	113
Giffin, Daniel B.	199	Shapiro, Jonathan S.	213
Girod, Lewis	283	Shapiro, Michael W.	15
Gutttag, John	269	Sinha, Anshumal	213
Hölzle, Urs	87	Srinivasan, Sudarshan M.	29
Kaashoek, M. Frans	199	Sripanidkulchai, Kunwadee	155
Kaminsky, Michael	199	Stathopoulos, Thanos	283
Kandula, Srikanth	29	Tan, Godfrey	269
Krishnamurthy, Arvind	99	Tracey, John M.	59
Krohn, Maxwell	185	Twining, Dan	45
Kubiatowicz, John	127	Voelker, Geoffrey M.	87
Kulkarni, Purushottam	59	Wang, Limin	171
Lai, Junwen	99	Wang, Randolph	99
LaVoie, Jason	59	Williamson, Matthew M.	45
Leventhal, Adam H.	15	Zhan, Jibin	155
Mazières, David	199	Zhang, Hui	141, 155
Mowbray, Miranda J.F.	45	Zhang, Ming	99
Ng, T.S. Eugene	141, 155	Zhou, Yuanyuan	29
Pai, Vivek	1, 171	Zwaenepoel, Willy	241
Pang, Ruoming	171		

Message from the Program Chairs

Welcome to the 2004 USENIX Annual Technical Conference!

It is a record-breaking year at USENIX: 164 papers were submitted to the General Track, an all-time high! Thank you for all of your submissions—it is of course those submissions that keep the quality of USENIX so high. Of those 164, the program committee chose 21 papers for presentation in the conference (just 12.8%). Perhaps a reflection of the times, the number of papers that were about distributed systems and networking was noticeably higher than in years past.

The review cycle consisted of three distinct phases. In the first phase, each committee member was assigned roughly 30 papers to review. In some cases, external reviewers were sought out, and the result was at least 3 reviews for each submitted paper. In the second phase, electronic discussion among reviewers was used to narrow down the pool of possible accepts, reducing the pool from 164 down to roughly 80 papers. Finally, in the third phase, we all huddled together in record-cold temperatures in Madison, Wisconsin, to decide on the final program of 21 papers. After a 12-hour day, with many tough choices made, our task was complete.

We also performed some simple analyses of paper submissions and acceptances. We found that submissions could be divided into roughly four quartiles by submission time. The first quarter of papers were submitted sometime before the last 8 hours until the deadline; the second quarter in the last 8 hours but not the last hour; the third quarter within the last hour but not the last 10 minutes; and finally the last quarter of papers (41!) were submitted within the last 10 minutes before the submission deadline. Interestingly (or perhaps randomly), the acceptance ratio was highest in the third quartile, where almost 25% of those papers got into the conference—so get your papers done a little (but not too) early!

Of course, there are many people to thank, all of whom were integral to the process of putting this program together. We would personally like to thank the program committee for their excellent efforts, Brian Noble for sharing his chairing wisdom and encouragement, Tim Denehy for his assistance in taking notes during the meeting, and most of all, the entire USENIX staff for all of their help throughout—without them, the process would have been many times more difficult and not nearly as enjoyable. A special thanks to Ellie Young and Jane-Ellen Long from USENIX for everything they have done to make our lives as co-chairs easy and fun.

We hope you enjoy this diverse and high-quality program! It is your participation that keeps the USENIX tradition alive.

Andrea Arpaci-Dusseau, *University of Wisconsin, Madison*
Remzi Arpaci-Dusseau, *University of Wisconsin, Madison*
Program Chairs

Making the “Box” Transparent: System Call Performance as a First-class Result

Yaoping Ruan and Vivek Pai
Department of Computer Science
Princeton University
{yruan,vivek}@cs.princeton.edu

Abstract

For operating system intensive applications, the ability of designers to understand system call performance behavior is essential to achieving high performance. Conventional performance tools, such as monitoring tools and profilers, collect and present their information off-line or via out-of-band channels. We believe that making this information *first-class* and exposing it to applications via *in-band* channels on a *per-call* basis presents opportunities for performance analysis and tuning not available via other mechanisms. Furthermore, our approach provides direct feedback to applications on time spent in the kernel, resource contention, and time spent blocked, allowing them to immediately observe how their actions affect kernel behavior. Not only does this approach provide greater *transparency* into the workings of the kernel, but it also allows applications to control how performance information is collected, filtered, and correlated with application-level events.

To demonstrate the power of this approach, we show that our implementation, DeBox, obtains precise information about OS behavior at low cost, and that it can be used in debugging and tuning application performance on complex workloads. In particular, we focus on the industry-standard SpecWeb99 benchmark running on the Flash Web Server. Using DeBox, we are able to diagnose a series of problematic interactions between the server and the OS. Addressing these issues as well as other optimization opportunities generates an overall factor of four improvement in our SpecWeb99 score, throughput gains on other benchmarks, and latency reductions ranging from a factor of 4 to 47.

1 Introduction

Operating system performance continues to be an active area of research, especially as demanding applications test OS scalability and performance limits. The kernel-user boundary becomes critically important as these applications spend a significant fraction, often a majority, of their time executing system calls. In the past, developers could expect to put data-sharing services, such as NFS, into the kernel to avoid the limitations stemming from running in user space. However, with the rapid rate of developments in HTTP servers, Web proxy servers, peer-to-peer systems,

and other networked systems, using kernel integration to avoid performance problems becomes unrealistic. As a result, examining the interaction between operating systems and user processes remains a useful area of investigation.

Much of the earlier work focusing on the kernel-user interface centered around developing new system calls that are more closely tailored to the needs of particular applications. In particular, zero-copy I/O [17, 31] and scalable event delivery [9, 10, 23] are examples of techniques that have been adopted in mainstream operating systems, via calls such as `sendfile()`, `transmitfile()`, `kevent()`, and `epoll()`, to address performance issues for servers. Other approaches, such as allowing processes to declare their intentions to the OS [32], have also been proposed and implemented. Some system calls, such as `madvise()`, provide usage hints to the OS, but with operating systems free to ignore such requests or restrict them to mapped files, programs cannot rely on their behavior.

Some recent research uses the reverse approach, where applications determine how the “black box” OS is likely to behave and then adapt accordingly. For example, the Flash Web Server [30] uses the `mincore()` system call to determine memory residency of pages, and combines this information with some heuristics to avoid blocking. The “gray box” approach [7, 15] manages to infer memory residency by observing page faults and correlating them with known replacement algorithms. In both systems, memory-resident files are treated differently than others, improving performance, latency, or both. These approaches depend on the quality of the information they can obtain from the operating system and the accuracy of their heuristics. As workload complexity increases, we believe that such inferences will become harder to make.

To remedy these problems, we propose a much more direct approach to making the OS transparent: make system call performance information a *first-class* result, and return it *in-band*. In practice, what this entails is having each system call fill a “performance result” structure, providing information about what occurred in processing the call. The term *first-class result* specifies that it gets treated the same as other results, such as `errno` and the system call return value, instead of having to be explicitly requested via other

system or library calls. The term *in-band* specifies that it is returned to the caller immediately, instead of being logged or processed by some other monitoring processes. While it is much larger and more detailed than the `errno` global variable, they are conceptually similar. Simply monitoring at the system call boundary, the scheduler, page fault handlers, and function entry and exit is sufficient to provide detailed information about the inner working of the operating system. This approach not only eliminates guesswork about what happens during call processing, but also gives the application control over how this information is collected, filtered, and analyzed, providing more customizable and narrowly-targeted performance debugging than is available in existing tools.

We evaluate the flexibility and performance of our implementation, DeBox, running on the FreeBSD operating system. DeBox allows us to determine where applications spend their time inside the kernel, what causes them to lose performance, what resources are under contention, and how the kernel behavior changes with the workload. The flexibility of DeBox allows us to measure very specific information, such as the kernel CPU consumption caused by a single call site in a program.

Our throughput experiments focus on analyzing and optimizing the performance of the Flash Web Server on the industry-standard SpecWeb99 benchmark [39]. Using DeBox, we are able to diagnose a series of problematic interactions between the server and the operating system on this benchmark. The resulting system shows an overall factor of four improvement in SpecWeb99 score, throughput gains on other benchmarks, and latency reductions ranging from a factor of 4 to 47. Most of the issues are addressed by application redesign and the resulting system is portable, as we demonstrate by showing improvements on Linux. Our kernel modifications, optimizing of the `sendfile()` system call, have been integrated into FreeBSD.

DeBox is specifically designed for performance analysis of the interactions between the OS and applications, especially in server-style environments with complex workloads. Its combination of features and flexibility is novel, and differentiates it from other profiling-related approaches. However, it is not designed to be a general-purpose profiler, since it currently does not address applications that spend most of their time in user space or in the “bottom half” (interrupt-driven) portion of the kernel.

The rest of this paper is organized as follows. In Section 2 we motivate the approach of DeBox. The detailed design and implementation are described in Section 3. We describe experimental setup and workloads in Section 4, then show a case study using DeBox to analyze and optimize the Flash Web Server in Section 5. Section 6 contains further experiments on latency and Section 7 demonstrates the portability of our optimizations. We discuss related work in Section 8 and conclude in Section 9.

2 Design Philosophy

DeBox is designed to bridge the divide in performance analysis across the kernel and user boundary by exposing kernel performance behavior to user processes, with a focus on server-style applications with demanding workloads. In these environments, performance problems can occur on either side of the boundary, and limiting analysis to only one side potentially eliminates useful information.

We present our observations about performance analysis for server applications as below. While some of these measurements could be made in other ways, we believe that DeBox’s approach is particularly well-suited for these environments. Note that replacing any of the existing tools is an explicit non-goal of DeBox, nor do we believe that such a goal is even feasible.

High overheads hide bottlenecks. The cost of the debugging tools may artificially stress parts of the system, thus masking the real bottleneck at higher load levels. Problems that appear only at high request rates may not appear when a profiler causes an overall slowdown. Our tests show that for server workloads, kernel `gprof` has 40% performance degradation even when low resolution profiling is configured. Others tracing and event logging tools generate large quantities of data, up to 0.5MB/s in Linux Trace Toolkit [42]. For more demanding workloads, the CPU or filesystem effects of these tools may be problematic.

We design DeBox not only to exploit hardware performance counters to reduce overhead, but also to allow users to specify the level of detail to control the overall costs. Furthermore, by splitting the profiling policy and mechanism in DeBox, applications can decide how much effort to expend on collecting and storing information. Thus, they may selectively process the data, discard redundant or trivial information, and store only useful results to reduce the costs. Not only does this approach make the cost of profiling controllable, but one process desiring profiling does not affect the behavior of others on the system. It affects only its own share of system resources.

User-level timing can be misleading. Figure 1 shows user-level timing measurement of the `sendfile()` system call in an event-driven server. This server uses nonblocking sockets and invokes `sendfile` only for in-memory data. As a result, the high peaks on this graph are troubling, since they suggest the server is blocking. A similar measurement using `getrusage()` also falsely implies the same. Even though the measurement calls immediately precede and follow the system call, heavy system activity causes the scheduler to preempt the process in that small window.

In DeBox, we integrate measurement into the system call process, so it does not suffer from scheduler-induced measurement errors. The DeBox-derived measurements of the same call are shown in Figure 2, and do not indicate such sharp peaks and blocking. Summary data for `sendfile` and `accept` (in non-blocking mode) are shown in Table 1.

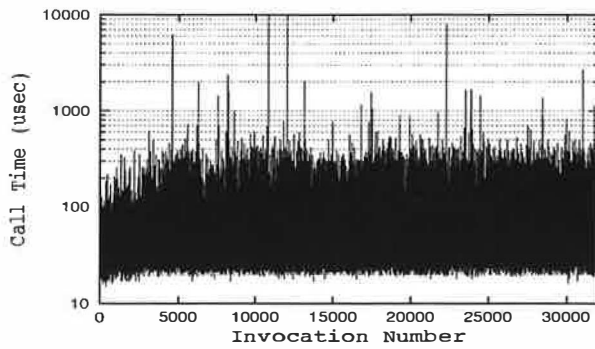


Figure 1: User-space timing of the `sendfile` call on a server running the SpecWeb99 benchmark – note the sharp peaks, which may indicate anomalous behavior in the kernel.

	<code>accept()</code>		<code>sendfile()</code>	
	User	DeBox	User	DeBox
Min	5.0	5.0	8.0	6.0
Median	10.0	6.0	60.0	53.0
Mean	14.8	10.5	86.6	77.5
Max	5216.0	174.0	12952.0	998.0

Table 1: Execution time (in usec) of two system calls measured in user application and DeBox – Note the large difference in maximums stemming from the measuring technique.

Statistical methods miss infrequent events. Profilers and monitoring tools may only sample events, with the belief that any event of interest is likely to take “enough” time to eventually be sampled. However, the correlation between frequency and importance may not always hold. Our experiments with the Flash web server indicate that adding a 1 ms delay to one out of every 1000 requests can degrade latency by a factor of 8 while showing little impact on throughput. This is precisely the kind of behavior that statistical profilers are likely to miss.

We eliminate this gap by allowing applications to examine every system call. Applications can implement their own sampling policy, controlling overhead while still capturing the details of interest to them.

Data aggregation hides anomalies. Whole-system profiling and logging tools may aggregate data to keep completeness and reduce overhead at the same time. This approach makes it hard to determine which call invocation experienced problems, or sometimes even which process or call site was responsible for high-overhead calls. This problem gets worse in network server environments where the systems are complex and large quantities of data are generated. It is not uncommon for these applications to have dozens of system call sites and thousands of invocations per second. For example, the Flash server consists of about 40 system calls and 150 calling sites. In these conditions, either discarding call history or logging full events is infeasible.

By making performance information a result of system calls, developers have control over how the kernel profiling is performed. Information can be recorded by process

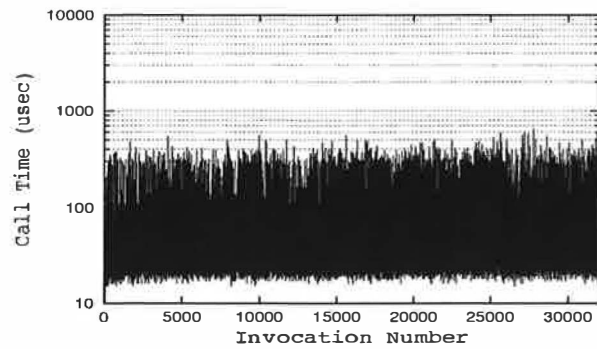


Figure 2: The same system call measured using DeBox shows much less variation in behavior.

and by call site, instead of being aggregated by call number inside the kernel. Users may choose to save accumulated results, record per-call performance history over time, or fully store some of the anomalous call trace.

Out-of-band reporting misses useful opportunities. As the kernel-user boundary becomes a significant issue for demanding applications, understanding the interaction between operating systems and user processes becomes essential. Most existing tools provide measurements out-of-band, making online data processing harder and possibly missing useful opportunities. For example, the online method allows an application to `abort()` or record the status when a performance anomaly occurs, but it is impossible with out-of-band reporting.

When applications receive performance information tied to each system call via in-band channels, they can choose the filtering and aggregation appropriate for the program’s context. They can easily correlate information about system calls with the underlying actions that invoke them.

3 Design & Implementation

This section describes our DeBox prototype implementation in FreeBSD and measures its overhead. We first describe the user-visible portion of DeBox, and then the kernel modifications. We compare overhead for DeBox support and active use versus an unmodified kernel. Examples of how to fully use DeBox and what kinds of information it provides are deferred to the case study in Section 5.

3.1 User-Visible Portion

The programmer-visible interface of DeBox is intentionally simple, since it consists of some monitoring data structures and a new system call to enable and disable data gathering. Figure 3 shows `DeBoxInfo`, the data structure that handles the DeBox information. It serves as the “performance information” counterpart to other system call results like `errno`. Programs wishing to use DeBox need to perform two actions: declare one or more of these structures as global variables, and call `DeBoxControl` to specify how much per-call performance information it desires.

```

typedef struct PerSleepInfo {
    int numSleeps;           /* # sleeps for the same reason */
    struct timeval blockedTime; /* how long the process is blocked */
    char wmesg[8];           /* reason for sleep (resource label) */
    char blockingFile[32];   /* file name causing the sleep */
    int blockingLine;        /* line number causing the sleep */
    int numWaitersEntry;     /* # of contenders at sleep */
    int numWaitersExit;      /* # of contenders at wake-up */
} PerSleepInfo;

typedef struct CallTrace {
    unsigned long callSite;   /* address of the caller */
    int deltaTime;           /* elapsed time in timer or CPU counter */
} CallTrace;

typedef struct DeBoxInfo {
    int syscallNum;           /* which system call */
    union CallTime {
        struct timeval callTimeval;
        long callCycles;     /* wall-clock time of entire call */
    } CallTime;
    int numPGFaults;          /* # page faults */
    int numPerSleepInfo;      /* # of filled PerSleepInfo elements */
    int traceDepth;           /* # functions called in this system call */
    struct PerSleepInfo psi[5]; /* sleeping info for this call */
    struct CallTrace ct[200];  /* call trace info for this call */
} DeBoxInfo;

int DeBoxControl(DeBoxInfo *resultBuf, int maxSleeps, int maxTrace);

```

Figure 3: DeBox data structures and function prototype

At first glance, the DeBoxInfo structure appears very large, which would normally be an issue since its size could affect system call performance. This structure size is not a significant concern, since the process specifies limits on how much of it is used. Most of the space is consumed by two arrays, PerSleepInfo and CallTrace. The PerSleepInfo array contains information about each of the times the system call blocks (sleeps) in the course of processing. The CallTrace array provides the history of what functions were called and how much time was spent in each. Both arrays are generously sized, and we do not expect many calls to fully utilize either one.

DeBoxControl can be called multiple times over the course of a process execution for a variety of reasons. Programmers may wish to have several DeBoxInfo structures and use different structures for different purposes. They can also vary the number of PerSleepInfo and CallTrace items recorded for each call, to vary the level of detail generated. Finally, they can specify a NULL value for resultBuf, which deactivates DeBox monitoring for the process.

3.2 In-Kernel Implementation

The kernel support for DeBox consists of performing the necessary bookkeeping to gather the data in the DeBoxInfo structure. The points of interest are system call entry and exit, scheduler sleep and wakeup routines, and function entry and exit for all functions reachable from a system call.

Since DeBox returns performance information when each system call finishes, the system call entry and exit code is modified to detect if a process is using DeBox. Once a process calls DeBoxControl and specifies how much of the arrays to use, the kernel stores this information and allocates a kernel-space DeBoxInfo reachable from the pro-

cess control block. This copy records information while the system call executes, avoiding many small copies between kernel and user. Prior to system call return, the requested information is copied back to user space.

At system call entry, all non-array fields of the process's DeBoxInfo are cleared. Arrays do not need to be explicitly cleared since the counters indicating their utilization have been cleared. Call number and start time are stored in the entry. We measure time using the CPU cycle counter available on our hardware, but we could also use timer interrupts or other facilities provided by the hardware.

Page faults that occur during the system call are counted by modifying the page fault handler to check for DeBox activation. We currently do not provide more detailed information on where faults occur, largely because we have not observed a real need for it. However, since the DeBoxInfo structure can contain other arrays, more detailed page fault information can be added if desired.

The most detailed accounting in DeBoxInfo revolves around the “sleeps”, when the system call blocks waiting on some resource. When this occurs in FreeBSD, the system call invokes the `tsleep()` function, which passes control to the scheduler. When the resource becomes available, the `wakeup()` function is invoked and the affected processes are unblocked. Kernel routines invoking `tsleep()` provide a human-readable label for use in utilities like `top`. We define a new macro for `tsleep()` in the kernel header files that permits us to intercept any sleep points. When this occurs, we record in a PerSleepInfo element where the sleep occurred (blockingFile and blockingLine), what time it started, what resource label was involved (wmesg), and the number of other processes waiting on the same resource (numWaitersEntry). Similarly, we modify the `wakeup()`


```

DeBoxInfo:
    4, /* system call # */
    3591064, /* call time, microseconds */
    989, /* # of page faults */
    2, /* # of PerSleepInfo used */
    0, /* # of CallTrace used (disabled) */

PerSleepInfo[0]:
    1270 /* # occurrences */
    723903 /* time blocked, microseconds */
    biowr /* resource label */
    kern/vfs_bio.c /* file where blocked */
    2727 /* line where blocked */
    1 /* # processes on entry */
    0 /* # processes on exit */

PerSleepInfo[1]:
    325 /* # occurrences */
    2710256 /* time blocked, microseconds */
    spread /* resource label */
    miscfs/specfs/spec_vnops.c /* file where blocked */
    729 /* line where blocked */
    1 /* # processes on entry */
    0 /* # processes on exit */

```

Figure 4: Sample DeBox output showing the system call performance of copying a 10MB mapped file

routine to provide `numWaitersExit` and calculate how much time was spent blocked. If the system call sleeps more than once at the same location, that information is aggregated into a single `PerSleepInfo` entry.

The process of tracing which kernel functions are called during a system call is slightly more involved, largely to minimize overhead. Conceptually, all that has to occur is that every function entry and exit point has to record the current time and function name when it started and finished, similar to what call graph profilers use. The gcc compiler allows entry and exit functions to be specified via the “instrument functions” option, but these are invoked by explicit function calls. As a result, function call overhead increases by roughly a factor of three. Our current solution involves manually inserting entry and exit macros into reachable functions. The entry macro pushes current function address and time in a temporary stack. The exit macro pops out the function address, calculates the wall clock time, and records these information in the `CallTrace` array. Automating this modification process should be possible in the future, and we are investigating using the `mcount()` kernel function used for kernel profiling.

To show what kind of information is provided in DeBox, we give a sample output in Figure 4. We memory-map a 10MB file, and use the `write()` system call to copy its contents to another file. The main `DeBoxInfo` structure shows that system call 4 (`write()`) was invoked, and it used about 3.6 seconds of wall-clock time. It incurred 989 page faults, and blocked in two unique places in the kernel. The first `PerSleepInfo` element shows that it blocked 1270 times at line 2727 in `vfs_bio.c` on “`biowr`”, which is the block I/O write routine. The second location was line 729 of `spec_vnops.c`, which caused 325 blocks at “`spread`”, read of a special file. The writes blocked for roughly 0.7 seconds, and the reads for 2.7 seconds.

3.3 Overhead

For DeBox to be attractive, it should generate low kernel overhead, especially in the common case. To quantify this overhead, we compare an unmodified kernel, a kernel with DeBox support, and the modified kernel with DeBox activated. We show these measurements in Table 2. The first column indicates the various system calls – `getpid()`, `gettimeofday()`, and `pread()` with various sizes. The second column indicates the time required for these calls on an unmodified system. The remaining columns indicate the additional overhead for various DeBox features.

call name or read size	base time	DeBox without call trace		DeBox call trace	
		off	on	off	on
<code>getpid</code>	0.46	+0.00	+0.50	+0.03	+1.45
<code>gettimeofday</code>	5.07	+0.00	+0.43	+0.03	+1.52
<code>pread 128B</code>	3.27	+0.02	+0.56	+0.21	+2.03
256 bytes	3.83	+0.00	+0.59	+0.26	+2.02
512 bytes	4.70	+0.00	+0.69	+0.28	+2.02
1024 bytes	6.74	+0.00	+0.68	+0.27	+2.02
2048 bytes	10.58	+0.03	+0.68	+0.26	+2.01
4096 bytes	18.43	+0.03	+0.74	+0.29	+2.16

Table 2: DeBox microbenchmark overheads – Base time uses an unmodified system. All times are in microseconds

We separate the measurement for call history tracing since we do not expect it will be activated continuously. These numbers show that the cost to support most DeBox features is minimal, and the cost of using the measurement infrastructure is tolerable. Since these costs are borne only by the applications that choose to enable DeBox, the overhead to the whole system is even lower. The performance impact with DeBox disabled, indicated by the 3rd column, is virtually unnoticeable. The cost of supporting call tracing, shown in the 5th column, where every function entry and exit point is affected, is higher, averaging approximately 5% of the system call time. This overhead is higher than ideal, and may not be desirable to have continuously enabled. However, our implementation is admittedly crude, and better compiler support could integrate it with the function prologue and epilogue code. We expect that we can reduce this overhead, along with the overhead of using the call tracing, with optimization.

	tar-gz a directory with		make kernel
	1MB file	10MB file	
base time	275.61 ms	3078.50 ms	236.96 s
basic on	+0.97 ms	+22.73 ms	+1.74 s
full support	+1.03 ms	+44.58 ms	+7.49 s

Table 3: DeBox macrobenchmark overheads

Since microbenchmarks do not indicate what kinds of slowdowns may be typically observed, we provide some macrobenchmark results to give some insight into these costs in Table 3. The three systems tested are: an unmodi-

fied system, one with only “basic” DeBox without call trace support, and one with complete DeBox support. The first two columns are times for archiving and compressing files of different sizes. The last column is for building the kernel. The overheads of DeBox support range from less than 1% to roughly 3% in the kernel build. We expect that many environments will tolerate this overhead in exchange for the flexibility provided by DeBox.

4 Experimental Setup & Workload

We describe our experimental setup and the relevant software components of the system in this section. All of our experiments, except for the portability measurements¹, are performed on a uniprocessor server running FreeBSD 4.6, with a 933MHz Pentium III, 1GB of memory, one 5400 RPM Maxtor IDE disk, and a single Netgear GA621 gigabit ethernet network adapter. The clients consist of ten Pentium II machines running at 300 MHz connected to a switch using Fast Ethernet. All machines are configured to use the default (1500 byte) MTU as required by SpecWeb99.

Our main application is the event-driven Flash Web Server, although we also perform some tests on the widely-used multi-process Apache [6] server. The Flash Web Server consists of a main process and a number of helper processes. The main process multiplexes all client connections, is intended to be nonblocking, and is expected to serve all requests only from memory. The helpers load disk data and metadata into memory to allow the main process to avoid blocking on disk. The number of main processes in the system is generally equal to the number of physical processors, while the number of helper processes is dynamically adjusted based on load. In previous tests, the Flash Web Server has been shown to compare favorably to high-performance commercial Web servers [30]. We run with logging disabled to simplify comparison with Apache, where enabling logging degrades performance noticeably.

We focus on the SpecWeb99 benchmark, an industry-standard test of the overall scalability of Web servers under realistic conditions. It is designed by SPEC, the developers of the widely-used SpecCPU workloads [38], and is based on traffic at production Web sites. Although not common in academia, it is the *de facto* standard in industry [27], with over 190 published results, and is different from most other Web server benchmarks in its complexity and requirements. It measures scalability by reporting the number of simultaneous connections the server can handle while meeting a specified quality of service. The data set and working set sizes increase with the number of simultaneous connections, and quickly exceed the physical memory of commodity systems. 70% of the requests are for static content, with the other 30% for dynamic content, including a mix of HTTP GET and POST requests. 0.15% of the requests require the use of a CGI process that must be spawned separately for each request.

¹The Linux kernel crashes on our existing server hardware

5 Case Study

In this section, we demonstrate how DeBox’s features can be used to analyze and optimize the behavior of the Flash Web Server. We discover a series of problematic interactions, trace their causes, and find appropriate solutions to avoid them or fix them. In the process, we gain insights into the causes of performance problems and how conventional solutions, such as throwing more resources at the problem, may exacerbate the problem. Our optimizations quadruple our SpecWeb99 score and also sharply decrease latency.

5.1 Initial experiments

Our first run of SpecWeb99 on the publicly available version of the Flash Web Server yields a SpecWeb99 result of roughly 200 simultaneous connections, much lower than the published score of 575 achieved on comparable hardware using TUX, an in-kernel Linux-only HTTP server. At 200 simultaneous connections, the dataset size is roughly 770MB, which is smaller than the amount of physical memory in the machine. Not surprisingly, the workload is CPU-bound, and a quick examination shows that the `mincore()` system call is consuming more resources than any other call in Flash.

The underlying problem is the use of linked lists in the FreeBSD virtual memory subsystem for handling virtual memory objects. The heavy use of memory-mapped files in Flash generates large numbers of memory objects, and a linear walk utilized by `mincore()` generates significant overhead. We apply a patch from Alan Cox of Rice University that replaces the linked lists with splay trees, and this brings `mincore()` in line with other calls. Our SpecWeb99 score rises to roughly 320, a 60% improvement. At this point, the working set has increased to 1.13GB, slightly exceeding our physical memory.

Once the `mincore()` problem is addressed, we still appear to be CPU-bound, and suspect the data copying is the bottleneck. So we update the Flash server to use the zero-copy I/O system call, `sendfile()`. However, using `sendfile()` requires that file descriptors be kept open, greatly increasing the number of file descriptors in use by Flash. To mitigate this impact, we implement support for `sendfile()` concurrently with support for `kevent()`, which is a scalable event delivery mechanism recently incorporated into FreeBSD. After these changes, we are not surprised by the drop in CPU utilization, but are surprised that the SpecWeb99 score drops to 300.

5.2 Successive refinement of detail

With the server exhibiting idle CPU time but an inability to meet SpecWeb99’s quality-of-service requirements, an obvious candidate is blocking system calls. However, Flash’s main process is designed to avoid blocking. We tried tracing the problem using existing tools, but found they suffered from the problems discussed in section 2. These experiences motivated the creation of DeBox.

biord/166	inode/127	getblk/1	sfpsy/1
<i>open</i> /162	<i>readlink</i> /84	<i>close</i> /1	<i>sendfile</i> /1
<i>read</i> /3	<i>open</i> /28		
<i>unlink</i> /1	<i>read</i> /9		
	<i>stat</i> /6		

Table 4: Summarized DeBox output showing blocking counts – The layout is organized by resource label and system call name. For example, of the 127 times this test blocked with the “inode” label, 28 were from the `open()` system call

The DeBox structures provide various levels of detail, allowing applications to specify what to measure. A typical use would first collect the basic DeBoxInfo to observe anomalies, then enable more details to identify the affected system calls, invocations, and finally the whole call trace.

We first use DeBox to get the blocking information, which is stored in the PerSleepInfo field. The PerSleepInfo data shows seven different system calls blocking in the kernel, and examination of the resource labels (wmesg) shows four reasons for blocking. These results are shown in Table 4, where each column header shows the resource label causing the blocking, followed by the total number of times blocked at that label. The elements in the column are the system calls that block on that resource, and the number of invocations involved. As evidenced by the calls involved, the “biord” (block I/O read) and “inode” (vnode lock) labels are both involved in opening and retrieving files, which is not surprising since our data set exceeds the physical memory of the machine.

The finest-grained kernel information is provided in the CallTrace structure, and we enable this level of detail once the PerSleepInfo identifies possible candidates. The main process should only be accessing cached data, so the fact that it blocks on disk-related calls is puzzling. For portability, the main process in Flash uses the helpers to demand-fetch disk data and metadata into the OS caches, and repeats the operation immediately after the helpers have completed loading, assuming that the recently loaded information will prevent it from blocking. Observing the full CallTrace of some of these blocking invocations shows the blocking is not caused by disk access, but contention on filesystem locks. Combining the blocking information from helper processes reveals that when the main process blocks, the helpers are operating on similarly-named files as the main process. We solve this problem by having the helpers return open file descriptors using `sendmsg()`, eliminating duplication of work in the main process. With this change, we are able to handle 370 simultaneous connections from SpecWeb99, with a dataset size of 1.28GB.

5.3 Capturing rare anomaly paths

We find that the `sendmsg()` change solves most of the filesystem-related blocking. However, one `open()` call in Flash still shows occasional blocking at the label “biord” (reading a disk block), but only after the server has been running for some time and under heavy workloads. Only

revealing which call induced the problem may not suffice a complete picture, because the reason of invoking that call is unclear. In a system with multiple identical system calls, existing tools do not have an efficient way to find which one causes the problem and the calling path involved.

Because DeBox information is returned in-band, the user-space context is also available once kernel performance anomaly is detected. On finding a blocking invocation of `open()`, we capture the path through the user code by calling `abort()` and using `gdb` to dump the stack². This approach uncovers a subtle performance bug in Flash induced by mapped-file cache replacement. Flash has two independent caches – one for URL-to-filename translations (name cache), and another for memory-mapped regions (data cache). For this workload, the name cache does not suffer from capacity misses, while the data cache may evict the least recently used entries. Under heavy load, a name cache hit and a data cache capacity miss causes Flash to erroneously believe that it had just recently performed the name translation and has the metadata cached. When Flash calls `open()` to access the file in this circumstance, the metadata associated with the name conversion is missing, causing blocking. We solve this problem by allowing the second set of helpers, the read helpers, to return file descriptors if the main process does not already have them open. After fixing this bug, we are able to handle 390 simultaneous connections, and a 1.34GB dataset.

5.4 Tracking call histories

With all blocking eliminated and with a much higher request rate, we return to the issue of CPU consumption. By storing the CallTime field of each system call, we can track per-call performance by invocation, both to observe trends and to identify time-related problems. Traditional profiling tools usually report average CPU consumption of each function, thus hiding any performance trends. User space timing functions may catch the general trend in spite of the measurement error, but involve much more work to track each system call and find the problematic ones.

5.4.1 Process creation overhead

By recording all CPU time values, we find that the largest call times are for the `fork()` system call and that its cost grows with the number of invocations, approaching 130 msec. Figure 5 shows the per-call time as a function of invocation. We observe that `fork()` time increases as the program runs, starting as low as 0.3 msec. These calls stem from the SpecWeb99 workload’s requirement that 0.15% of the requests be handled by forking new processes.

A full call trace indicates that `fork()` spends the bulk of its time copying file descriptors and VM map entries (for mapped regions). Rather than changing the implementation of `fork()`, we opt to slightly modify the Flash archi-

²Alternately, we could invoke `fork()` followed by `abort()` to keep the process running while still obtaining a snapshot, or we could record the call path manually

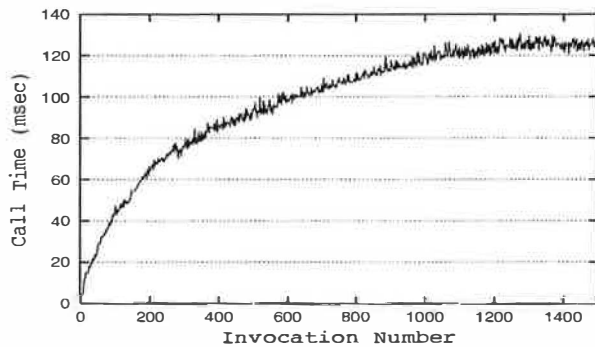


Figure 5: Call time of `fork()` as a function of invocation

ture. We introduce a new helper process that is responsible for creating the CGI processes. Since this new process does not map files or cache open files, its `fork()` time is not affected by the main process size. This change yields a 10% improvement, to 440 simultaneous connections and a 1.50GB dataset size.

5.4.2 Memory lookup overhead

Though the dataset size now exceeds physical memory by over 50%, the system bottleneck remains CPU. Examining the time consumption of each system call again reveals that most time is being spent in memory residency checking. Though our modified Flash uses `sendfile()`, it uses `mincore()` to determine memory residency, which requires that files be memory-mapped. The cumulative overhead of memory-map operations is the largest consumer of CPU time. As can be seen in Figure 6, the per-call overhead of `mmap()` is significant and increases as the server runs. The cost increase is presumably due to finding available space as the process memory map becomes fragmented.

To avoid the memory-residency overheads, we use Flash’s mapped-file cache bookkeeping as the sole heuristic for guessing memory residency. We eliminate all `mmap`, `mincore`, and `munmap` calls but keep track of what pieces of files have been recently accessed. Sizing the cache conservatively with respect to main memory, we save CPU overhead but introduce a small risk of having the main process block. The CPU savings of this change is substantial, allowing us to reach 620 connections (2GB dataset).

5.5 Profiling by call site

We take advantage of DeBox’s flexibility by separating the kernel time consumption based on call site rather than call name. We are interested in the cost of handling dynamic content since SpecWeb99 includes 30% dynamic requests which could be processed by various interfaces. Flash uses a persistent CGI interface similar to FastCGI [28] to reuse CGI processes when possible, and this mechanism communicates over pipes. Although the `read()` and `write()` system calls are used by the main process, the helpers, and all of the CGI processes, we measure the overhead of only those involved in communication with CGI processes.

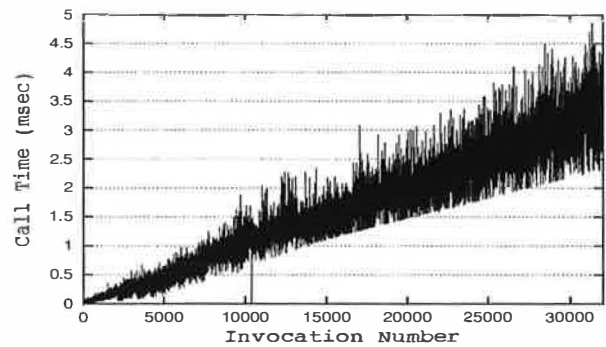


Figure 6: Call time of `mmap()` as a function of invocation

Our measurements show that the single call site responsible for most of the time is where the main process reads from the CGIs, consuming 20% of all kernel time, (176 seconds out of 891 seconds total). Writing the request to the CGI processes is much smaller, requiring only 24.3 seconds of system call time. This level of detail demonstrates the power of making performance a first-class result, since existing kernel profilers would not have been able to separate the time for the `read()` calls by call sites. By modifying our CGI interface slightly, the main process writes only the HTTP header to the client, and passes the socket to the CGI application to let it write the data directly. This change allows us to reach 710 connections (2.35GB dataset).

5.6 Other optimization opportunities

By replacing our exact memory residency check with a cheaper heuristic, we gain performance, but introduce blocking into the `sendfile()` system call. New `PerSleepInfo` measurements of the blocking behavior of `sendfile()` are shown in Table 5.

time	label	kernel file	line
6492	sfbufa	kern/uipc.syscalls.c	1459
702	getblk	kern/kern_lock.c	182
984544	biord	kern/vfs_bio.c	2724

Table 5: New blocking measurements of `sendfile()`

The resource label “sfbufa” indicates that the kernel has exhausted the `sendfile` buffers used to map filesystem pages into kernel virtual memory. We confirm that increasing the number of buffers during boot time may mitigate this problem in our test. However, based on the results of previous copy-avoidance systems [17, 31], we opt instead to implement recycling of kernel virtual address buffers. With this change, many requests to the same file do not cause multiple mappings, and eliminates the associated virtual memory and physical map (`pmap`) operations. Caching these mappings may temporarily use more wired memory than no caching, but the reduction in overhead and address space consumption outweighs the drawbacks.

The other two resource labels, “getblk” and “biord”, are related to disk access initiated within `sendfile()` when the requested pages are not in memory. Even though the

socket being used is nonblocking, that behavior is limited only to network buffer usage. We introduce a new flag to `sendfile()` so that it returns a different `errno` value if disk blocking would occur. This change allows us to achieve the same effect as we had with `mincore()`, but with much less CPU overhead. We may optionally have the read helper process send data directly back to the client on a filesystem cache miss, but have not implemented this optimization.

However, even with blocking eliminated, we find performance barely changes when using `sendfile()` versus `writew()`, and we find that the problem stems from handling small writes. HTTP responses consist of a small header followed by file data. The `writew()` code aggregates the header and the first portion of the body data into one packet, benefiting small file transfers. In SpecWeb99, 35% of all static requests are for files 1KB or smaller.

The FreeBSD `sendfile()` call includes parameters specifying headers and trailers to be sent with the data, whereas the Linux implementation does not. Linux introduces a new socket option, `TCP_CORK`, to delay transmission until full packets can be assembled. While FreeBSD's "monolithic" approach provides enough information to avoid sending a separate header, its implementation uses a kernel version of `writew()` for the header, thus generating an extra packet. We improve this implementation by creating an mbuf chain using the header and body data before sending it to lower levels of the network stack. This change generates fewer packets, improving performance and network latency. Results of these changes on a microbenchmark are shown in Figure 7. With the `sendfile()` changes, we are able to achieve a SpecWeb99 score of 820, with a dataset size of 2.7GB.

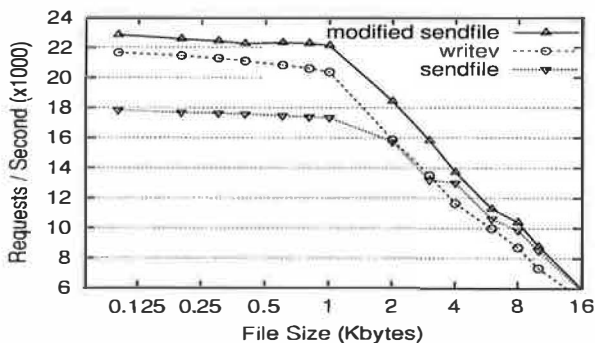


Figure 7: Microbenchmark performance comparison of `writew`, `sendfile`, and `modified sendfile` – In this test, all clients request a single file at full speed using persistent connections.

5.7 Case Study Summary

By addressing the interaction areas identified by DeBox, we achieve a factor of four improvement in our SpecWeb99 score, supporting four times as many simultaneous connections while also handling a data set that almost three times as large as the physical memory of our machine. The

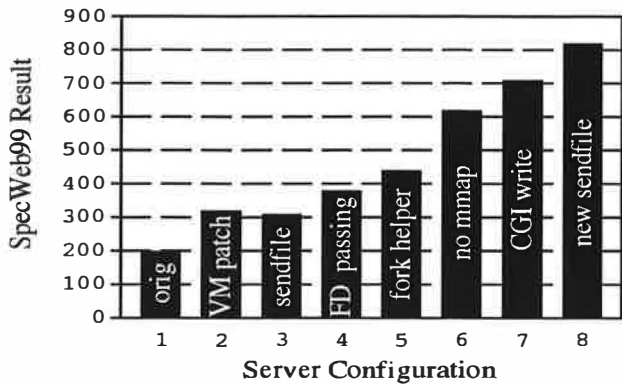


Figure 8: SpecWeb99 summary – 1. Original 2. VM patch 3. Using `sendfile()` 4. FD-passing helpers 5. Fork helper 6. Eliminate mmap 7. New CGI interface 8. New `sendfile()`

SpecWeb99 results of our modifications can be seen in Figure 8, where we show the scores for all of the intermediate modifications we made. Our final result of 820 compares favorably to published SpecWeb99 scores, though no directly comparable systems have been benchmarked. We outperform all uniprocessor systems with similar memory configurations but using other server software – the highest score for a system with less than 2GB of memory is 575.

Most of our changes are portable architectural modifications to the Flash Web Server, including (1) passing file descriptors between the helpers and the main process to avoid most disk operations in the main process, (2) introducing a new `fork()` helper to handle forking CGI requests, (3) eliminating the mapped file cache, and (4) allowing CGI processes to write directly to the clients instead of writing to the main process. Figure 9 shows the original and new architectures of the static content path for the server.

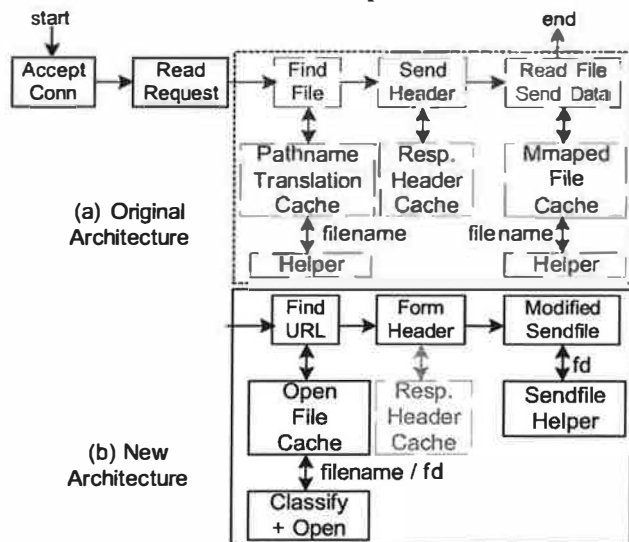


Figure 9: Architectural changes – The architecture is greatly simplified by using file descriptor passing and eliminating mapped file caching. Modified components are indicated with dark boxes.

The changes we make to the operating system focus on `sendfile()`, including (1) adding a new flag and return

value to indicate when blocking on disk would occur, (2) caching kernel address space mapping to avoid unnecessary physical map operations, and (3) sending headers and file data in a single mbuf chain to avoid multiple packets for small responses. Additionally, we apply a virtual memory system patch that ultimately is superfluous since we remove the memory-mapped file cache. We have provided our modifications to the FreeBSD developer group and all three optimizations have been incorporated into FreeBSD.

6 Latency

Since we identify and correct many sources of blocking, we are interested in the effects of our changes on server latency. We first compare the effect of our changes on the SpecWeb99 workload, and then reproduce workloads used by other researchers in studying static content latencies. In all cases, we compare latencies using a workload below the maximum of the slowest server configuration under test.

6.1 SpecWeb99 workload

On the SpecWeb99 workload, we find that mean response time is reduced by a factor of four by our changes. The cumulative distribution of latencies can be seen in Figure 10. We use 300 simultaneous connections, and compare the new server with the original Flash running on a patched VM system. Since 30% of the requests are for longer-running dynamic content, we also test the latencies of a SpecWeb99 test with only static requests. The mean of this workload is 7.1 msec, lower than the 10.6 msec mean for the new server running the complete workload. This difference suggests that further optimization of dynamic content handling may lead to even better performance. To compare the difference between static and dynamic request handling, we calculate the 5th, 50th, and 95th percentiles of the latencies for requests on the SpecWeb99 workload. These results are shown in Table 6, and indicate that dynamic content is served at roughly half the speed of its static counterpart. The latency difference between the new server and the original Flash on this test is not as large as expected because the working set still fits in physical memory.

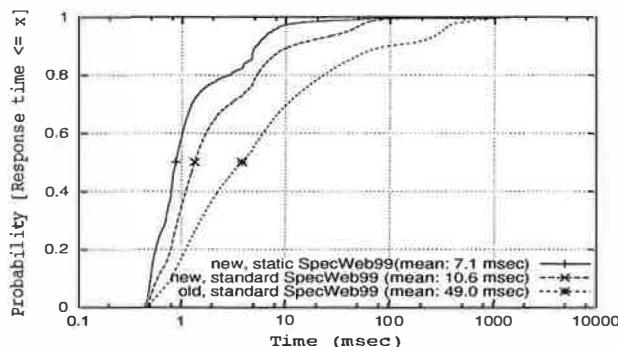


Figure 10: Latency summary for 300 SpecWeb99 connections

	5%(ms)	50%(ms)	95%(ms)	mean(ms)
static	0.51	1.45	59.81	9.92
dynamic	0.99	2.83	91.31	12.19

Table 6: Separating SpecWeb99 static and dynamic latencies

6.2 Diskbound static workload

To determine our latency benefit on a more disk-bound workload and to compare our results with those of other researchers, we construct a static workload similar to the one used to evaluate the Haboob server [41]. In this workload, 1020 simulated clients generate static requests to a 3.3GB data set. Persistent connections are used, with clients issuing 5 requests per connection before closing it. To avoid overload, the request rate is fixed at 2300 requests/second, which is roughly 90% of the slowest server's capacity.

We compare several configurations to determine the latency benefits and the impact of parallelism in the server. We run the new and original versions of Flash with a single instance and four instances, to compare uniprocessor configurations with what would be expected on a 4-way SMP. We also run Apache with 150 and 300 server processes.

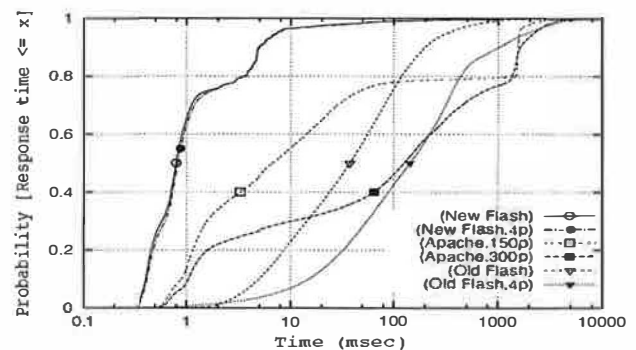


Figure 11: Response latencies for the 3.3GB static workload

	5% (ms)	median (ms)	95% (ms)	mean (ms)
New Flash	0.37	0.79	7.45	7.56
New Flash, 4p	0.38	0.82	7.51	7.72
Old Flash	3.36	37.59	326.40	92.37
Old Flash, 4p	7.05	142.65	1924.42	420.85
Apache 150p	0.70	6.64	1599.50	360.62
Apache 300p	0.78	124.98	2201.63	545.93

Table 7: Summaries of the static workload latencies

The results, given in Figure 11 and Table 7, show the response time of our new server under this workload exhibits improvements of more than a factor of twelve in mean response time, and a factor of 47 in median latency. With four instances, the differences are a factor of 54 in mean response time and 174 in median time. We measure the maximum capacities of the servers when run in infinite-demand mode, and these results are shown in Table 8. While the throughput gain from our optimizations is significant, the scale of gain is much lower than the SpecWeb99 test, in-

data set	Apache	Old Flash	New Flash
500MB	240.3	485.2	660.9
1.5GB	230.7	410.6	580.3
3.3GB	210.6	264.5	326.4

Table 8: Server static workload capacities (Mb/s)

dicating that our latency benefits do not stem purely from extra capacity.

6.3 Excess parallelism

We also observe that all servers tested show latency degradation when running with more processes, though the effect is much lower for our new server. This observation is in line with the self-interference between the helpers and the main Flash process which we described earlier. We increase the number of helper processes and measure its effect on the SpecWeb99 results, as shown in Table 9. We observe that too few helpers is insufficient to fully utilize the disk, and increasing their number initially helps performance. However, the blocking from self-interference increases, eventually decreasing performance. A similar phenomenon, stemming from the same problem, is also observed with Apache. Using DeBox, we find that Apache with 150 processes, sleeps 3667 times per second, increasing to 3994 times per second at 300 processes. This behavior is responsible for Apache's latency increase in Figure 11.

# of helpers	1	5	10	15
Blocking count	114	295	339	394
% Conforming	40.9%	95.1%	96.9%	89.5%

Table 9: Parallelism benefits and self-interference – The conformance measurement indicates how many requests meet SpecWeb99's quality-of-service requirement.

This result suggests that excess parallelism, where server designers use parallelism for convenience, may actually degrade performance noticeably. This observation may explain the latency behavior reported for Haboob [41].

7 Results Portability

The main goal of this work is to provide developers with tools to diagnose and correct the performance problems in their own applications. Thus we hope that the optimizations made on one platform also have benefit on other platforms. To test this premise, we test the performance on Linux, which has no DeBox support.

Unfortunately, we were unable to get Linux to run properly on our existing hardware, despite several attempts to resolve the issue on the Linux kernel list. So, for these numbers, we use a server machine with a 3.0 GHz Pentium 4 processor and two Intel Pro1000/MT Gigabit adapters, 1GB of memory, and a similar disk. The experiments were performed on 2.4.21 kernel with `epoll()` support.

We compare the throughput and latency of four servers: Apache 1.3.27, Haboob, Flash, and the new Flash. We increase the max number of clients to 1024 in Apache and

disable logging. Both the original Flash and the new Flash server use the maximum available cache size for LRU. We also adjust the cache size in Haboob for the best performance. The throughput results, shown in Table 10, are quite surprising. The Haboob server, despite having aggressive optimizations and event-driven stages, performs slightly better than Apache on diskbound workload but worse than Apache on an in-memory workload. We believe that its dependence on excess parallelism (via its threaded design) may have some impact on its performance. The new Flash server gains about 17-24% over the old one for the smaller workloads, and all four servers have similar throughput on the larger workload because of diskbound.

Throughput (Mb/s)				
data set	Haboob	Apache	Flash	New Flash
500MB	324.9	434.3	1098.1	1284.7
1.5GB	303.4	372.4	661.7	822.5
3.3GB	184.1	177.4	173.8	199.1
Response Time (ms)				
profile	Haboob	Apache	Flash	New Flash
5%	78.2	0.22	0.21	0.15
median	414.3	0.61	1.56	0.42
95%	1918.9	661.8	412.5	3.68
mean	656.2	418.0	512.5	141.9

Table 10: Throughput measurement on Linux with 1GB memory

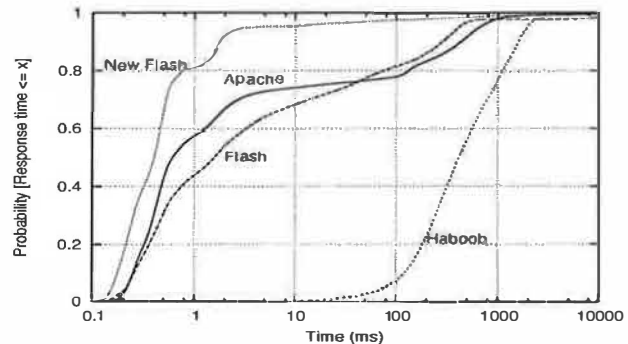


Figure 12: Response time on Linux with 3.3GB dataset

Despite similar throughputs at the 3.3GB data set size, the latencies of the servers, shown in Figure 12 and Table 10, are markedly different. The Haboob latency profile is very close to their published results, but are beaten by all of the other servers. We surmise that the minimal amount of tuning done to configurations of Apache and the original Flash yield much better results than the original Haboob comparison [41]. The benefit of our optimization is still valid on this platform, with a factor of 4 both in median and mean latency over the original Flash. One interesting observation is that the 95% latency of the new Flash is a factor of 39 lower than the mean value. This result suggests that the small fraction of long-latency requests is the major contribution to the mean latency. Though our Linux results are not directly comparable to our FreeBSD ones due

to the hardware differences, we do notice this phenomenon is less obvious on FreeBSD. Presumably one of the causes of this is the blocking disk I/O feature of `sendfile()` on Linux. Another reason may be Linux's filesystem performance, since this throughput is worse than what we observed on FreeBSD.

8 Related Work

To compare DeBox's approach of making performance information a first-class result, we describe three categories of tools currently in use, and explain how DeBox relates to these approaches.

- **Function-based profilers** – Programs such as `prof`, `gprof` [18], and their variants are often used to detect hot-spots in programs and kernels. These tools use compiler assistance to add bookkeeping information (count and time). Data is gathered while running and analyzed offline to reveal function call counts and CPU usage, often along edges in the call graph.
- **Coverage-based profilers** – These profilers divide the program of interest into regions and use a clock interrupt to periodically sample the location of the program counter. Like function-based profilers, data gathering is done online while analysis is performed offline. Tools such as `profil()`, `kernbb`, and `tcov` can then use this information to show what parts of the program are most likely to consume CPU time. Coverage-only approaches may miss infrequently-called functions entirely and may not be able to show call graph behavior. Coverage information combined with compiler analysis can be used to show usage on a basic-block basis.
- **Hardware-assisted profilers** – These profilers are similar to coverage-based profilers, but use special features of the microprocessor (event counters, timers, programmable interrupts) to obtain high-precision information at lower cost. The other major difference is that these profilers, such as DCPI [4], Morph [43], VTune [19], Oprofile [29], and PP[3] tend to be whole system profilers, capturing activity across all processes and the operating system.

In this category, DeBox is logically closest to kernel `gprof`, though it provides more than just timing information. DeBox's full call trace allows more complete call graph generation than `gprof`'s arc counts, and with the data compression and storage performed in user space, overhead is moved from the kernel to the process. Compared to path profiling, DeBox allows developers to customize the level of detail they want about specific paths, and it allows them to act on that information as it is generated. In comparison to low-level statistical profilers such as DCPI, coverage differs since DeBox measures functions directly used during the system call. As a result, the difference in approach yields some differences in what can be gathered and the difficulty in doing so – DCPI can gather bottom-half information, which DeBox currently cannot. However, DeBox can

easily isolate problematic paths and their call sites, which DCPI's aggregation makes more difficult.

- **System activity monitors** – Tools such as `top`, `vmstat`, `netstat`, `iostat`, and `systat` can be used to monitor a running system or determine a first-order cause for system slowdowns. The level of precision varies greatly, with `top` showing per-process information on CPU usage, memory consumption, ownership, and running time, to `vmstat` showing only summary information on memory usage, fault rates, disk activity, and CPU usage.
- **Trace tools** – Trace tools provide a means of observing the system call behavior of processes without access to source code. Tools such as `truss`, PCT [11], `strace` [2], and `ktrace` are able to show some details of system calls, such as parameters, return values, and timing information. Recent tools, such as `Kitrace` [21] and the Linux Trace Toolkit [42], also provide data on some kernel state that changes as a result of the system calls. These tools are intended for observing another process, and as a result, producing out-of-band measurements and data aggregation, often requiring post-processing to generate usable output.
- **Timing calls** – Using `gettimeofday()` or similar calls, programmers can manually record the start and end times of events to infer information based on the difference. The `getrusage()` call adds some information beyond timings (context switches, faults, messages and I/O counts) and can similarly be used. If per-call information is required, not only do these approaches introduce many more system calls, but the information can be misleading.

DeBox compares favorably with a hypothetical merger of the timing calls and the trace tools in the sense that timing information is presented in-band, but so is the other information. In comparison with the Linux Trace Toolkit, our focus differs in that we gather the most significant pieces of data related to performance, and we capture it at a much higher level of detail.

- **Microbenchmarks** – Tools such as `lmbench` [24] and `hbench:OS` [13] can measure best-case times or the isolated cost of certain operations (cache misses, context switches, etc.). Common usage for these tools is to compare different operating systems, different hardware platforms, or possible optimizations.
- **Latency tools** – Recent work on attempting to find the source of latency on desktop systems not designed for real-time work have yielded insight and some tools. The Intel Real-Time Performance Analyzer [33] helps automate the process of pinpointing latency. The work of Cota-Robles and Held [16] and Jones and Regehr [20] demonstrate the benefits of successive measurement and searching.
- **Instrumentation** – Dynamic instrumentation tools provide mechanisms to instrument running systems (processes or the kernel) under user control, and to obtain precise kernel information. Examples include `DynInst` [14], `KernInst` [40], `ParaDyn` [25], `Etch` [35], and `ATOM` [37].

The appeal of this approach versus standard profilers is the flexibility (arbitrary code can be inserted) and the cost (no overhead until use). Information is presented out-of-band.

Since DeBox measures the performance of calls in their natural usage, it resembles the instrumentation tools. DeBox gains some flexibility by presenting this data to the application, which can filter it on-line. One major difference between DeBox and kernel instrumentation is that we provide a rich set of measurements to any process, rather than providing information only to privileged processes.

Beyond these performance analysis tools, the idea of observing kernel behavior to improve performance has appeared in many different forms. We share similarities with Scheduler Activations [5] in observing scheduler activity to optimize application performance, and with the Infokernel system by Arpaci-Dusseau et al. [8]. Our goals differ, since we are more concerned with understanding why blocking occurs rather than reacting to it during a system call. Our non-blocking `sendfile()` modification is patterned on non-blocking sockets, but it could be used in other system calls as well. In a similar vein, RedHat has applied for a patent on a new flag to the `open()` call, which aborts if the necessary metadata is not in memory [26].

Our observations on blocking and its impact on latency may impact server design. Event-driven designs for network servers have been a popular approach since the performance studies of the Harvest Cache [12] and the Flash server [30]. Schmidt and Hu [36] performed much of the early work in studying threaded architectures for improving server performance. A hybrid architecture was used by Welsh et al. [41] to support scheduling, while Larus and Parkes [22] demonstrate that such scheduling can also be performed in event-driven architectures. Qie et al. [34] show that such architectures can also be protected against denial-of-service attacks. Adya et al. [1] discuss the unification of the two models. We believe that DeBox can be used to identify problem areas in other servers and architectures, as our latency measurements of Apache suggest.

9 Conclusions and Discussion

This paper presents the design, implementation and evaluation of DeBox, an effective approach to provide more OS transparency, by exposing system call performance as a first-class result via in-band channels. DeBox provides direct performance feedback from the kernel on a per-call basis, enabling programmers to diagnose kernel and user interactions correlated with user-level events. Furthermore, we believe that the ability to monitor behavior on-line provides programmatic flexibility of interpreting and analyzing data not present in other approaches.

Our case study using the Flash Web Server with the SpecWeb99 benchmark running on FreeBSD demonstrates the power of DeBox. Addressing the problematic interactions and optimization opportunities discovered using De-

Box improves our experimental results an overall factor of four in SpecWeb99 score, despite having a data set size nearly three times as large as our physical memory. Furthermore, our latency analysis demonstrates gains between a factor of 4 to 47 under various conditions. Further results show that fixing the bottlenecks identified using DeBox also mitigates most of the negative impact from excess parallelism in application design.

We have shown how DeBox can be used in a variety of examples, allowing developers to shape profiling policy and react to anomalies in ways that are not possible with other tools. Although DeBox does require access to kernel source code for achieving the highest impact, we do not believe that such a restriction is significant. FreeBSD, NetBSD, and Linux sources are easily available, and with the advent of Microsoft's Shared Source initiatives, few hardware platforms exist for which some OS source is not available. Also, general information about kernel behavior instead of source code may be enough to help application redesign. Our performance portability results also demonstrate that our new system achieves better performance even without kernel modification. A further implication of this is that it is possible to perform analysis and modifications while running on one operating system, and still achieve some degree of benefit in other environments.

In this paper we focused on how DeBox can be used as a performance analysis tool, but we have not discussed its utility in general-purpose monitoring because of space limits. Given its low overheads, DeBox is an excellent candidate for monitoring long-running applications. We are approaching this problem by modifying the `libc` library and associated header files so that a simple recompile and relink will enable monitoring of applications using DeBox. It is also possible to process results automatically by allowing user-specified analysis policies. We are working on such a tool, which will allow passive monitoring of daemons, but a full discussion of it is beyond the scope of this paper.

While we have shown DeBox to be effective in identifying performance problems in the interaction between the OS and applications, the current version of DeBox does not handle the bottom-half activities in the kernel. DeBox's current focus on the system call boundary also makes it less useful for tracing problems arising purely in user space. However, we believe that both of these limitations can be addressed, and we are continuing work in these areas.

Acknowledgments

This work was partially supported by an NSF CAREER Award. We would like to thank our shepherd, Remzi Arpaci-Dusseau, and our anonymous reviewers for their feedback and insight. We would also like to thank the FreeBSD developers, particularly Mike Silbersack and Alan Cox, for their efforts in integrating our `sendfile()` modifications.

References

- [1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative tasking without manual stack management. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.
- [2] W. Akkerman. strace. <http://www.wi.leidenuniv.nl/~wichert/strace/>.
- [3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, June 1997.
- [4] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Wehl. Continuous profiling: Where have all the cycles gone. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 1–14, Saint-Malo, France, Oct. 1997.
- [5] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.
- [6] Apache Software Foundation. The Apache Web server. <http://www.apache.org/>.
- [7] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. of the 19th ACM Symp. on Operating System Principles*, pages 43–56, Chateau Lake Louise, Banff, Canada, Oct. 2001.
- [8] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proc. of the 18th ACM Symp. on Operating System Principles*, pages 90–105, Bolton Landing, NY, Oct. 2003.
- [9] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.
- [10] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX 1999 Annual Technical Conference*, pages 253–265, Monterey, CA, June 1999.
- [11] C. Blake and S. Bauer. Simple and general statistical profiling with pct. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.
- [12] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125, 1995.
- [13] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *ACM SIGMETRICS Conference*, pages 214–224, Seattle, WA, June 1997.
- [14] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
- [15] N. Burnett, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.
- [16] E. Cota-Robles and J. P. Held. A comparison of windows driver model latency performance on windows NT and windows 98. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation*, pages 159–172, New Orleans, LA, Feb. 1999.
- [17] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the 14th ACM Symp. on Operating System Principles*, pages 189–202, Asheville, NC, Dec. 1993.
- [18] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, Boston, Massachusetts, June 1982.
- [19] Intel. Vtune Performance Analyzers Homepage. <http://developer.intel.com/software/products/vtune/index.htm>.
- [20] M. B. Jones and J. Regehr. The problems you're having may not be the problems you think you're having: Results from a latency study of windows nt. In *7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.
- [21] G. Kuenning. Kitrace—precise interactive measurement of operating systems kernels. *SOFTWARE—PRACTICE AND EXPERIENCE*, 1(1):1–21, 1994.
- [22] J. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *USENIX 2002 Annual Technical Conference*, pages 103–114, Monterey, CA, June 2002.
- [23] J. Lemon. Kqueue: A generic and scalable event notification facility. In *FREENIX Track: USENIX 2001 Annual Technical Conference*, pages 141–154, Boston, MA, June 2001.
- [24] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*, pages 279–294, San Diego, CA, June 1996.
- [25] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(1):37–46, 1995.
- [26] I. Molnar. Method and apparatus for atomic file look-up. United States Patent Application #20020059330, May 16, 2002.
- [27] E. Nahum. Deconstructing SPECweb99. In *7th International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, CO, Aug. 2002.
- [28] Open Market. FastCGI. <http://www.fastcgi.com/>.
- [29] OProfile. <http://oprofile.sourceforge.net/>.
- [30] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.
- [31] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.
- [32] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, 1993.
- [33] L. K. Puthiyedath, E. Cota-Robles, J. Keys, and J. P. H. Anil Aggarwal. The design and implementation of the intel real-time performance analyzer. In *Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, Sept. 2002.
- [34] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.
- [35] T. Romer, G. V. D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *USENIX Windows NT Workshop*, pages 1–8, 1997.
- [36] D. C. Schmidt and J. C. Hu. Developing flexible and high-performance Web servers with frameworks and patterns. *ACM Computing Surveys*, 32(1):39, 2000.
- [37] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.
- [38] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. <http://www.spec.org/cpu2000>.
- [39] Standard Performance Evaluation Corporation. SPEC Web 96 & 99 Benchmarks. <http://www.spec.org/osg/web96/> and <http://www.spec.org/osg/web99/>.
- [40] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation*, pages 117–130, New Orleans, LA, Feb. 1999.
- [41] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of the 19th ACM Symp. on Operating System Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, Oct. 2001.
- [42] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.
- [43] C. X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen, and M. D. Smith. System support for automated profiling and optimization. In *Proc. of the 16th ACM Symp. on Operating System Principles*, pages 15–26, Saint-Malo France, Oct. 1997.

Dynamic Instrumentation of Production Systems

Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal

Solaris Kernel Development

Sun Microsystems

{bmc, mws, ahl}@eng.sun.com

Abstract

This paper presents DTrace, a new facility for dynamic instrumentation of production systems. DTrace features the ability to dynamically instrument both user-level and kernel-level software in a unified and absolutely safe fashion. When not explicitly enabled, DTrace has zero probe effect — the system operates exactly as if DTrace were not present at all. DTrace allows for many tens of thousands of instrumentation points, with even the smallest of systems offering on the order of 30,000 such points in the kernel alone. We have developed a C-like high-level control language to describe the predicates and actions at a given point of instrumentation. The language features user-defined variables, including thread-local variables and associative arrays. To eliminate the need for most postprocessing, the facility features a scalable mechanism for aggregating data and a mechanism for speculative tracing. DTrace has been integrated into the Solaris operating system and has been used to find serious systemic performance problems on production systems — problems that could not be found using pre-existing facilities.

1 Introduction

As systems grow larger and more complicated, performance analysis is increasingly performed by the system integrator in production rather than by the developer in development. Trends towards componentization and application consolidation accelerate this change: system integrators increasingly combine off-the-shelf components in ways that the original developers did not anticipate. Performance analysis infrastructure has generally not kept pace with the shift to in-production performance analysis: the analysis infrastructure is still focussed on the developer, on development systems, or both. And where performance analysis infrastructure *is* designed for production use, it is almost always *process-centric* — and therefore of little help in understanding systemic problems.

To be acceptable for use on production systems, perfor-

mance analysis infrastructure must have zero probe effect when disabled, and must be absolutely safe when enabled. That is, its mere presence must not make the system any slower, and there must be no way to accidentally induce system failure through misuse. To have systemic scope, the entire system must be instrumentable, and there must exist ways to easily coalesce data to highlight systemic trends.

We have developed a facility for systemic dynamic instrumentation that can gather and coalesce arbitrary data on production systems. This facility — DTrace — has been integrated into Solaris and is publicly available[12]. DTrace features:

- **Dynamic instrumentation.** Static instrumentation always induces some disabled probe effect; to achieve the zero disabled probe effect required for production use, DTrace uses *only* dynamic instrumentation. When DTrace is not in use, the system is just as if DTrace were not present at all.
- **Unified instrumentation.** DTrace can dynamically instrument both user *and* kernel-level software, and can do so in a *unified* manner whereby both data and control flow can be followed across the user/kernel boundary.
- **Arbitrary-context kernel instrumentation.** DTrace can instrument virtually all of the kernel, including delicate subsystems like the scheduler and synchronization facilities.
- **Data integrity.** DTrace always reports any errors that prevent trace data from being recorded. In the absence of such errors, DTrace *guarantees* data integrity: there are no windows in which recorded data can be silently corrupted or lost.
- **Arbitrary actions.** The actions taken at a given point of instrumentation are not defined or limited *a priori* — the user can enable any probe with an arbitrary set of actions. Moreover, DTrace guarantees

absolute safety of user-defined actions: run-time errors such as illegal memory accesses are caught and reported.

- **Predicates.** A logical predicate mechanism allows actions to be taken only when user-specified conditions are met, thereby pruning unwanted data *at the source*. DTrace thus avoids retaining, copying and storing data that will ultimately be discarded.
- **A high-level control language.** Predicates and actions are described in a C-like language — dubbed “D” — that supports all ANSI C operators and allows access to the kernel’s variables and native types. D offers user-defined variables, including global variables, thread-local variables, and associative arrays. D also supports pointer dereferencing; coupled with the run-time safety mechanisms of DTrace, structure chains can be safely traversed in a predicate or action.
- **A scalable mechanism for aggregating data.** DTrace allows data to be aggregated based on an arbitrary tuple of D expressions. The mechanism coalesces data as it is generated, reducing the amount of data that percolates through the framework by a factor of the number of data points. By allowing aggregation based on D expressions, DTrace permits users to aggregate by virtually anything.
- **Speculative tracing.** DTrace has a mechanism for speculatively tracing data, deferring the decision to commit or discard the data to a later time. This feature eliminates the need for most post-processing when exploring sporadic aberrant behavior.
- **Heterogeneous instrumentation.** Tracing frameworks have historically been designed around a single instrumentation methodology. In DTrace, the instrumentation providers are formally separated from the probe processing framework by a well-specified API, allowing novel dynamic instrumentation technologies to plug into and exploit the common framework.
- **Scalable architecture.** DTrace allows for many tens of thousands of instrumentation points (even the smallest systems typically have on the order of 30,000 such points) and provides primitives for subsets of probes to be efficiently selected and enabled.
- **Virtualized consumers.** Everything about DTrace is virtualized per consumer: multiple consumers can enable the same probe in different ways, and a single consumer can enable a single probe in different

ways. There is no limit on the number of concurrent DTrace consumers.

The remainder of this paper describes DTrace in detail. In Section 2, we discuss related work in the area of dynamic instrumentation. Section 3 provides an overview of the DTrace architecture. Section 4 describes some of the instrumentation providers we have implemented for DTrace. Section 5 describes the D language. Section 6 describes the DTrace facility for aggregating data. Section 7 describes the user-level instrumentation provided by DTrace. Section 8 describes the DTrace facility for speculative tracing. Section 9 describes in detail a production performance problem that was root-caused using DTrace. Finally, Section 10 discusses future work and Section 11 provides our conclusions.

2 Related work

The notion of safely augmenting operating system execution with user-specified code has been explored in extensible systems like VINO[10] and SPIN[2]. More generally, the notion of augmenting execution with code has been explored in aspect-oriented programming systems like AspectJ[6]. However, these systems were designed to allow the user to *extend* the system or application where DTrace is designed to allow the user to simply *understand* it. So where the extensible systems allow much more general purpose augmentation, they have many fewer (if any) primitives for understanding system behavior.

Systems like ATOM[11] and Purify[3] instrument programs for purposes of understanding them, but these systems fundamentally differ from DTrace in that they are *static* — they operate by instrumenting the binary off-line and running the instrumented binary in lieu of the original binary. Moreover, these static instrumentation systems don’t provide systemic insight: they cannot integrate instrumentation from disjoint applications, and they are generally unable to instrument the operating system whatsoever. Even in the confines of their domain of single application instrumentation, these systems are inappropriate for production environments: in these environments, application restart represents an unacceptable lapse in service.

There is a large body of work dedicated to systemic and dynamic instrumentation. Some features of DTrace, like predicates, were directly inspired by other work[8]. Some other features, like the idea of a higher-level language for system monitoring, exist elsewhere[1, 4, 9] — but DTrace has made important new contributions like thread-local variables and associative arrays. Other fea-

tures, like aggregations, exist only in rudimentary form elsewhere[1, 4]; DTrace has advanced these ideas significantly. And some features, like speculative tracing, don't seem to exist in any form in any of the prior work.

2.1 Linux Trace Toolkit

The Linux Trace Toolkit (LTT) is designed around a traditional static instrumentation methodology that induces a non-zero (but small) probe effect for each instrumentation point[16]. To keep the overall disabled probe effect reasonably low, LTT defines only a limited number of instrumentation points — comprising approximately 45 events. LTT cannot take arbitrary actions (each statically-defined event defines an event-specific “detail”), and lacks any sort of higher-level language to describe such actions. LTT has a coarse mechanism for pruning data, whereby traced events may be limited only to those pertaining to a given PID, process group, GID or UID, but no other predicates are possible. As LTT has few mechanisms for reducing the data flow via pruning or coalescing, substantial effort has naturally gone into optimizing the path of trace data from the kernel to user-level[17].

2.2 DProbes

DProbes is a facility originally designed for OS/2 that was ported to Linux and subsequently expanded[9]. Superficially, DProbes and DTrace have some similar attributes: both are based on dynamic instrumentation (and thus both have zero probe effect when not enabled) and both define a language for arbitrary actions as well as a simple virtual machine to implement them. However, there are significant differences. While DProbes uses dynamic instrumentation, it uses a technique that is lossy when a probe is hit simultaneously on different CPUs. While DProbes has user-defined variables, it lacks thread-local variables and associative arrays. Further, it lacks any mechanism for data aggregation, and has no predicate support. And while DProbes has made some safety considerations (for example, invalid loads are handled through an exception mechanism), it was not designed with absolute safety as a constraint; misuse of DProbes can result in a system crash.¹

2.3 K42

K42 is a research kernel that has its own static instrumentation framework[14]. K42's instrumentation has many of LTT's limitations (statically defined actions, no

¹Examples of such misuse include erroneously specifying a non-instruction boundary to instrument or specifying an action that incorrectly changes register values.

facilities for data reduction, etc.), but — as in DTrace — thought has been given in K42 to instrumentation scalability. Like DTrace, K42 has lock-free, per-CPU buffering — but K42 implements it in a way that sacrifices the integrity of traced data.² Recently, the scalable tracing techniques from K42 have been integrated into LTT, presumably rectifying LTT's serious scalability problems (albeit at the expense of data integrity).

2.4 Kerninst

Kerninst is a dynamic instrumentation framework that is designed for use on commodity operating system kernels[13]. Kerninst achieves zero probe effect when disabled, and allows instrumentation of virtually any text in the kernel. However, Kerninst is highly aggressive in its instrumentation; users can erroneously induce a fatal error by accidentally instrumenting routines that are not actually safe to instrument.³ Kerninst allows for some coalescence of data, but data may not be aggregated based on arbitrary tuples. Kerninst has some predicate support, but it does not allow for arbitrary predicates and has no support for arbitrary actions.

3 DTrace Architecture

The core of DTrace — including all instrumentation, probe processing and buffering — resides in the kernel. Processes become DTrace *consumers* by initiating communication with the in-kernel DTrace component via the DTrace library. While any program may be a DTrace consumer, `dtrace(1M)` is the canonical DTrace consumer: it allows generalized access to all DTrace facilities.

3.1 Providers and Probes

The DTrace framework itself performs no instrumentation of the system; that task is delegated to instrumentation *providers*. Providers are loadable kernel modules that communicate with the DTrace kernel module using a well-defined API. When they are instructed to do so by the DTrace framework, instrumentation providers determine points that they can potentially instrument. For every point of instrumentation, providers call back into the DTrace framework to create a *probe*. To create a probe the provider specifies the module name and function name of the instrumentation point, plus a semantic

²For example, rescheduling during data recording can silently corrupt the data buffer.

³In particular, Kerninst on SPARC makes no attempt to recognize text as being executed at `TL=1` or `TL>1` — two highly constrained contexts in the SPARC V9 architecture. Instrumenting such text with Kerninst induces an operating system panic. This has been communicated to Miller et al.; a solution is likely forthcoming[7].

name for the probe. Each probe is thus uniquely identified by a 4-tuple:

< provider, module, function, name >

Probe creation does *not* instrument the system: it simply identifies a potential for instrumentation to the DTrace framework. When a provider creates a probe, DTrace returns a *probe identifier* to the provider.

Probes are advertised to consumers, who can enable them by specifying any (or all) elements of the 4-tuple. When a probe is enabled, an *enabling control block* (ECB) is created and associated with the probe. If there are no other ECBs associated with the probe (that is, if the probe is disabled), the DTrace framework calls the probe's provider to enable the probe. The provider dynamically instruments the system in such a way that when the probe fires, control is transferred to an entry point in the DTrace framework with the probe's identifier specified as the first argument. A key attribute of DTrace is that there are no constraints as to the context of a firing probe: the DTrace framework itself is non-blocking and makes no explicit or implicit calls into the kernel at-large.

When a probe fires and control is transferred to the DTrace framework, interrupts are disabled on the current CPU, and DTrace performs the activities specified by each ECB on the probe's ECB chain. Interrupts are then reenabled and control returns to the provider. The provider itself need not handle any multiplexing of consumers on a single probe — all multiplexing is handled by the framework's ECB abstraction.

3.2 Actions and Predicates

Each ECB may have an optional predicate associated with it. If an ECB has a predicate and the condition specified by the predicate is not satisfied, processing advances to the next ECB. Every ECB has a list of actions; if the predicate is satisfied, the ECB is processed by iterating over its actions. If an action indicates data to be recorded, the data is stored in the per-CPU buffer associated with the consumer that created the ECB; see Section 3.3. Actions may also update D variable state; user variables are described in more detail in Section 5. Actions may *not* store to kernel memory, modify registers, or make otherwise arbitrary changes to system state.⁴

⁴There do exist some actions that change the state of the system, but they change state only in a well-defined way (e.g. stopping the current process, or inducing a kernel breakpoint). These destructive actions are only permitted to users with sufficient privilege, and can be disabled entirely.

3.3 Buffers

Each DTrace consumer has a set of in-kernel per-CPU buffers allocated on its behalf and referred to by its consumer state. The consumer state is in turn referred to by each of the consumer's ECBs; when an ECB action indicates data to be recorded, it is recorded in the ECB consumer's per-CPU buffer. The amount of data recorded by a given ECB is always *constant*. That is, different ECBs may record different amounts of data, but a given ECB always records the same quantity of data. Before processing an ECB, the per-CPU buffer is checked for sufficient space; if there is not sufficient space for the ECB's data recording actions, a per-buffer *drop count* is incremented and processing advances to the next ECB.

It is up to consumers to minimize drop counts by reading buffers periodically.⁵ Buffers are read out of the kernel using a mechanism that both maintains data integrity and assures that probe processing remains wait-free. This is done by having two per-CPU buffers: an active buffer and an inactive buffer. When a DTrace consumer wishes to read the buffer for a specified CPU, a cross-call is made to the CPU. The cross-call, which executes on the specified CPU, disables interrupts on the CPU, switches the active buffer with the inactive buffer, reenables interrupts and returns. Because interrupts are disabled in both probe processing and buffer switching (and because buffer switching always occurs on the CPU to be switched), an ordering is assured: buffer switching and probe processing cannot possibly interleave on the same CPU. Once the active and inactive buffers have been switched, the inactive buffer is copied out to the consumer.

The data record layout in the per-CPU buffer is an *enabled probe identifier* (EPID) followed by some amount of data. An EPID has a one-to-one mapping with an ECB, and can be used to query the kernel for the size and layout of the data stored by the corresponding ECB. Because the data layout for a given ECB is guaranteed to be constant over the lifetime of the ECB, the ECB metadata can be cached at user-level. This design separates the metadata stream from the data stream, simplifying run-time analysis tools considerably.

3.4 DIF

Actions and predicates are specified in a virtual machine instruction set that is emulated in the kernel at probe firing time. The instruction set, "D Intermediate Format" or DIF, is a small RISC instruction set designed for simple emulation and on-the-fly code generation. It

⁵Consumers may also reduce drops by increasing the size of in-kernel buffers.

features 64-bit registers, 64-bit arithmetic and logical instructions, comparison and branch instructions, 1-, 2-, 4- and 8-byte memory loads from kernel and user space, and special instructions to access variables and strings. DIF is designed for simplicity of emulation. For example, there is only one addressing mode and most instructions operate only on register operands.

3.5 DIF Safety

As DIF is emulated in the context of a firing probe, it is a design constraint that DIF emulation be absolutely safe. To assure basic sanity, opcodes, reserved bits, registers, string references and variable references are checked for validity as the DIF is loaded into the kernel. To prevent DIF from inducing an infinite loop in probe context, only *forward* branches are permitted. This safety provision may seem draconian — it eliminates loops altogether — but in practice we have not discovered it to present a serious limitation.⁶

Run-time errors like illegal loads or division by zero cannot be detected statically; these errors are handled by the DIF virtual machine. Misaligned loads and division by zero are easily handled — the emulator simply refuses to perform such operations. (Any attempt to perform such an operation aborts processing of the current ECB and results in a run-time error that is propagated back to the DTrace consumer.) Similarly, loads from memory-mapped I/O devices (where loads may have undesirable or dangerous side effects) are prevented by checking that the address of a DIF-directed load does not fall within the virtual address range that the kernel reserves for memory-mapped device registers.

Loads from unmapped memory are more complicated to prevent, however, because it is not possible to probe VM data structures from probe firing context. When the emulation engine attempts to perform such a load, a hardware fault will occur. The kernel's page fault handler has been modified to check if the load is DIF-directed; if it is, the fault handler sets a per-CPU bit to indicate that a fault has occurred, and increments the instruction pointer past the faulting load. After emulating each load, the DIF emulation engine checks for the presence of the faulted bit; if it is set, processing of the current ECB is aborted and the error is reported to the user. This mechanism adds some processing cost to the kernel's page fault path, but the cost is so small relative to the total processing cost of a page fault that the effect on system performance is nil.

⁶DProbes addressed this problem by allowing loops but introducing a user-tunable, “`jmpmax`,” as an upper-bound on the number of jumps that a probe handler may make.

4 Providers

By formally separating instrumentation providers from the core framework, DTrace is able to accommodate heterogeneous instrumentation methodologies. Further, as future instrumentation methodologies are developed, they can be easily plugged in to the DTrace framework. We have implemented twelve different instrumentation providers that between them offer observability into many aspects of the system. While the providers employ different instrumentation methodologies, *all* of the DTrace providers have no observable probe effect when disabled. Some of the providers are introduced below, but the details of their instrumentation methodologies are largely beyond the scope of this paper.

4.1 Function Boundary Tracing

The Function Boundary Tracing (FBT) provider makes available a probe upon entry to and return from nearly every function in the kernel. As there are many functions in the kernel, FBT provides many probes — even on the smallest systems, FBT will provide more than 25,000 probes. As with other DTrace providers, FBT has zero probe effect when it is not explicitly enabled, and when enabled only induces a probe effect in probed functions. While the mechanism used for the implementation of FBT is highly specific to the instruction set architecture, FBT has been implemented on both SPARC and x86.

On SPARC, FBT works by replacing an instruction with an unconditional annulled branch-always (ba,a) instruction. The branch redirects control flow into an FBT-controlled trampoline, which prepares arguments and transfers control into DTrace. Upon return from DTrace, the replaced instruction is executed in the trampoline before transferring control back to the instrumented code path. This is a similar mechanism to that used by Kerninst[13] — but it is at once less general (it instruments only function entry and return) and completely safe (it will never erroneously instrument code executed at $TL > 0$).

On x86, FBT uses a trap-based mechanism that replaces one of the instructions in the sequence that establishes a stack frame (or one of the instructions in the sequence that dismantles a stack frame) with an instruction to transfer control to the interrupt descriptor table (IDT). The IDT handler uses the trapping instruction pointer to look up the FBT probe and transfers control into DTrace. Upon return from DTrace, the replaced instruction is *emulated* from the trap handler by manipulating the trap stack. The use of emulation (instead of instruction rewriting and reexecution) assures that FBT does

not suffer from the potential lossiness of the DProbes mechanism.

4.2 Statically-defined Tracing

While FBT allows for comprehensive probe coverage, one must be familiar with the kernel implementation to use it effectively. To have probes with semantic meaning, one must allow probes to be statically declared in the implementation. The mechanism for implementing this is typically a macro that expands to a conditional call into a tracing framework if tracing is enabled[16]. While the probe effect of this mechanism is small, it is observable: even when disabled, the expanded macro induces a load, a compare and a taken branch.

In keeping with our philosophy of zero probe effect when disabled, we have implemented a statically-defined tracing (SDT) provider by defining a C macro that expands to a call to a non-existent function with a well-defined prefix (“_dtrace_probe_”). When the kernel linker sees a relocation against a function with this prefix, it replaces the call instruction with a no-operation and records the full name of the bogus function along with the location of the call site. When the SDT provider loads, it queries this auxiliary structure and creates a probe with a name specified by the function name. When an SDT probe is enabled, the no-operation at the call site is patched to be a call into an SDT-controlled trampoline that transfers control into DTrace.

In principle, this provider induces a disabled probe effect: while the call site is replaced with a no-operation, the compiler must nonetheless treat the call site as a transfer of control into an unknown function. As a result, this mechanism can induce disabled probe effect by creating artificial register pressure. However, by carefully placing SDT probes near extant calls to functions, this disabled probe effect can be made so small as to be unobservable. Indeed, we have added over 150 such sites to the Solaris kernel, and have been unable to measure any performance difference — even on microbenchmarks designed to highlight any disabled probe effect.

4.3 Lock Tracing

The lockstat provider makes available probes that can be used to understand virtually any aspect of kernel synchronization behavior. The lockstat provider works by dynamically rewriting the kernel functions that manipulate synchronization primitives. As with all other DTrace providers, this instrumentation only occurs as probes are explicitly enabled; the lockstat provider induces zero probe effect when not enabled.

The lockstat provider’s instrumentation methodology has existed in Solaris for quite some time — it has historically been the basis for the lockstat(1M) command. As such, the lockstat provider is particularly useful for understanding kernel resource contention. As part of the DTrace work, the in-kernel component was augmented to become the lockstat provider, the lockstat command was reimplemented as a DTrace consumer, and the legacy custom-built, single-purpose data-processing framework was discarded.

4.4 System Call Tracing

The syscall provider makes available a probe at the entry to and return from each system call in the system. As system calls are the primary interface between user-level applications and the operating system kernel, the syscall provider can offer tremendous insight into application behavior with respect to the system. The syscall provider works by dynamically rewriting the corresponding entry in the system call table when a probe is enabled.

4.5 Profiling

The providers described above provide probes that are anchored to specific points in text. However, DTrace also allows for *unanchored probes* — probes that are not associated with any particular point of execution but rather with some asynchronous event source. Among these is the profile provider, for which the event source is a time-based interrupt of specified interval. These probes can be used to sample some aspect of system state every specified unit of time, and the samples can then be used to infer system behavior. Given the arbitrary actions that DTrace supports, the profile provider can be used to sample practically any datum in the system. For example, one could sample the state of the current thread, the state of the CPU, the current stack trace, or the current machine instruction.

5 D Language

DTrace users can specify arbitrary predicates and actions using the high-level D programming language. D is a C-like language that supports all ANSI C operators and allows access to the kernel’s native types and global variables. D includes support for several kinds of user-defined variables, including global, clause-local, and thread-local variables and associative arrays. D programs are compiled into DIF by a compiler implemented in the DTrace library; the DIF is then bundled into an in-memory object file representation and sent to the in-kernel DTrace framework for validation and probe en-

abling. The `dtrace(1M)` command provides a generic front-end to the D compiler and DTrace, but other layered tools can be built on top of the compiler library as well, such as the new implementation of `lockstat(1M)` described earlier.

5.1 Program Structure

A D program consists of one or more *clauses* that describe the instrumentation to be enabled by DTrace. Each probe clause has the form:

```
probe-descriptions
/predicate/
{
    action-statements
}
```

Probe descriptions are specified using the form *provider:module:function:name*. Omitted fields match any value, and `sh(1)` globbing syntax is supported. The predicate and action statements may each be optionally omitted.

D uses a program structure similar to `awk(1)` because tracing programs resemble pattern matching programs in that execution order does not follow traditional function-oriented program structure; instead, execution order is defined by a set of external inputs and the tracing program “reacts” by executing the predefined matching clauses. During internal testing, the meaning of this program form was immediately obvious to UNIX developers and permitted rapid adoption of the language.

5.2 Types, Operators and Expressions

As C is the language of UNIX, D is designed to form a companion language to C for use in dynamic tracing. D predicates and actions are written identically to C language statements, and all of the ANSI C operators can be used and follow identical precedence rules. D also supports all of the intrinsic C data types, `typedef`, and the ability to define `struct`, `union`, and `enum` types. Users are also permitted to define and manipulate their own variables, described shortly, and access a set of predefined functions and variables provided by DTrace.

The D compiler also makes use of C source type and symbol information provided by a special kernel service, allowing D programmers to access C types and global variables defined in kernel source code without declaring them. The FBT provider exports the input arguments and return values of kernel functions to DTrace when its probes fire, and the C type service also allows the D compiler to automatically associate these arguments

with their corresponding C data types in a D program clause that matches an FBT probe.

Unlike a traditional C source file, a D source file may access types and symbols from a variety of separate scopes, including the core kernel, multiple loadable kernel modules, and any type and variable definitions provided in the D program itself. To manage access to external namespaces, the backquote (‘) character can be inserted in symbol and type identifiers to reference the namespace denoted by the identifier preceding the backquote. For example, the type `struct foo`bar` would name the C type `struct bar` in a kernel module named `foo`, and the identifier ``rootvp` would match the kernel global variable `rootvp` and would have the type `vnode_t *` automatically assigned to it by the D compiler.

5.3 User Variables

D variables can be declared using C declaration syntax in the outer scope of the program, or they can be implicitly defined by assignment statements. When variables are defined by assignment, the left-hand identifier is defined and assigned the type of the right-hand expression for the remainder of the program. Our experience showed that D programs were rapidly developed and edited and often written directly on the `dtrace(1M)` command-line, so users benefited from the ability to omit declarations for simple programs.

5.4 Variable Scopes

In addition to global variables, D programs can create *clause-local* and *thread-local* variables of any type. Variables from these two scopes are accessed using the reserved prefixes `this->` and `self->` respectively. The prefixes serve to both separate the variable namespaces and to facilitate their use in assignment statements without the need for prior declaration. Clause-local variables access storage that is re-used across the execution of D program clauses, and are used like C automatic variables. Thread-local variables associate a single variable name with separate storage for each operating system thread, including interrupt threads.

Thread-local variables are used frequently in D to associate data with a thread performing some activity of interest. As an example, Figure 1 contains a script that uses thread-local variables to output the amount of time that a thread spends in a `read(2)` system call.

```

syscall::read:entry
{
    self->t = timestamp;
}

syscall::read:return
/self->t/
{
    printf("%d/%d spent %d nsecs in read\n",
        pid, tid, timestamp - self->t);
}

```

Figure 1: A script that uses a thread-local variable to output the amount of time that a thread spends in a read(2) system call. The thread-local variable `self->t` is instantiated on-demand when any thread fires the `syscall::read:entry` probe and is assigned the value of the built-in `timestamp` variable; the program then computes the time difference when the system call returns. As with recorded data, DTrace reports any failure to allocate a dynamic variable so data is never silently lost.

5.5 Associative Arrays

D programs can also create associative array variables where each array element is indexed by a tuple of expression values and data elements are created on-demand. For example, the D program statement `a[123, "hello"] = 456` defines an associative array `a` with tuple signature `[int, string]` where each element is an `int`, and then assigns the value 456 to the element indexed by tuple `[123, "hello"]`. D supports both global and thread-local associative arrays. As in other languages such as Perl, associative arrays permit D programmers to easily create and manage complex dictionary data structures without requiring them to manage memory and write lookup routines.

5.6 Strings

D provides a built-in `string` type to resolve the ambiguity of the C `char*` type, which can be used to represent an arbitrary address, the address of a single character, or the address of a NUL-terminated string. The D `string` type acts like the C type `char [n]` where `n` is a fixed string limit that can be adjusted at compile-time. The string limit is also enforced by the DTrace in-kernel component, so that it can provide built-in functions such as `strlen()` and ensure finite running time when an invalid string address is specified. D permits strings to be copied using the `=` operator and compared using the relational operators. D implicitly promotes `char*` and `char []` to `string` appropriately.

6 Aggregating Data

When instrumenting the system to answer performance-related questions, it is often useful to think not in terms of data gathered by individual probes, but rather how that data can be aggregated to answer a specific question. For example, if one wished to know the number of system calls by user ID, one would not necessarily care about the datum collected at *each* system call — one simply wants to see a table of user IDs and system calls. Historically, this question has been answered by gathering data at each system call, and postprocessing the data using a tool like `awk(1)` or `perl(1)`. However, in DTrace the aggregating of data is a first-class operation, performed *at the source*.

6.1 Aggregating Functions

We define an *aggregating function* to be one that has the following property:

$$f(f(x_0) \cup f(x_1) \cup \dots \cup f(x_n)) = f(x_0 \cup x_1 \cup \dots \cup x_n)$$

where x_n is a set of arbitrary data. That is, applying an aggregating function to subsets of the whole and then applying it again to the set of results gives the same result as applying it to the whole itself. Many common functions for understanding a set of data are aggregating functions, including counting the number of elements in the set, computing the maximum value of the set, and summing all elements in the set. Not all functions are aggregating functions, however; computing the mode and computing the median are two examples of non-aggregating functions.

Applying aggregating functions to data *in situ* has a number of advantages:

- The entire data set need not be stored. Whenever a new element is to be added to the set, the aggregating function is calculated given the set consisting of the current intermediate result and the new element. After the new result is calculated, the new element may be discarded. This reduces the amount of storage required by a factor of the number of data points — which is often quite large.
- A scalable implementation is allowed. One does not wish for data collection to induce pathological scalability problems. Aggregating functions allow for intermediate results to be kept *per-CPU* instead of in a shared data structure. When a system-wide result is desired, the aggregating function may then be applied to the set consisting of the per-CPU intermediate results.

6.2 Aggregations

DTrace implements aggregating functions as *aggregations*. An aggregation is a named structure indexed by an *n*-tuple that stores the result of an aggregating function. In D, the syntax for an aggregation is:

```
@identifier [keys] = aggfunc(args);
```

where *identifier* is an optional name of the aggregation, *keys* is a comma-separated list of D expressions, *aggfunc* is one of the DTrace aggregating functions and *args* is a comma-separated list of arguments to the aggregating function. (Most aggregating functions take just a single argument that represents the new datum.)

For example, the following DTrace script counts `write(2)` system calls by application name:

```
syscall::write:entry
{
    @counts[execname] = count();
}
```

By default, aggregation results are displayed when `dtrace(1M)` terminates. (This behavior may be changed by explicitly controlling aggregation output with the `printa` function.) Assuming the above were named “`write.d`”, running it might yield:

```
# dtrace -s write.d
dtrace: script 'write.d' matched 1 probe
^C
dtrace          1
cat              4
sed              9
head            9
grep            14
find            15
tail            25
mountd          28
expr            72
sh              291
tee             814
sshd            1996
make.bin        2010
```

In the above output, one might perhaps be interested in understanding more about the `write` system calls from the processes named “`sshd`.” For example, to get a feel for the distribution of write sizes per file descriptor, one could aggregate on `arg0` (the file descriptor argument to the `write` system call), specifying the “`quantize()`” aggregating function (which generates a power-of-two distribution) with an argument of `arg2` (the size argument to the `write` system call):

```
syscall::write:entry
/execname == "sshd"/
{
    @[arg0] = quantize(arg2);
}
```

Running the above yields a frequency distribution for each file descriptor. For example:

5	value	----- Distribution -----	count
	16		0
	32		1
	64		0
	128		0
	256		13
	512		13
	1024		199
	2048		0

The above output would indicate that for file descriptor five, 199 writes were between 1024 and 2047 bytes. If one wanted to understand the origin of writes to this file descriptor, one could (for example) add to the predicate that `arg0` be five, and aggregate on the application's stack trace by using the `ustack` function:

```
syscall::write:entry
/execname == "sshd" && arg0 == 5/
{
    @[ustack()] = quantize(arg2);
}
```

7 User-level Instrumentation

DTrace provides instrumentation of user-level program text through the `pid` provider, which can instrument arbitrary instructions in a specified process. The `pid` provider is slightly different from other providers in that it actually defines a *class* of providers — each process can potentially have an associated `pid` provider. The process identifier is appended to the name of each `pid` provider. For example, the probe `pid1203:libc.so.1:malloc:entry` corresponds to the function entry of `malloc(3C)` in process 1203.

In keeping with the DTrace philosophy of dynamic instrumentation, target processes need not be restarted to be instrumented and, as with other providers, there is no `pid` provider probe effect when the probes are not enabled.

The techniques used by the `pid` provider are ISA-specific, but they all involve a mechanism that rewrites the instrumented instruction to induce a trap into the operating system. The trap-based mechanism has a higher enabled probe effect than branch-based mechanisms used elsewhere[15], but it completely unifies kernel- and user-level instrumentation: any DTrace mechanism that may be used with kernel-level probes may also be used with user-level probes. As an example, Figure 2 contains a script that uses a thread-local D variable to follow *all* activity — user-level *and* kernel-level — from a specified user-level function; Figure 3 contains example output of this script.

```
#!/usr/sbin/dtrace -s

#pragma D option flowindent

pid$1::$2:entry
{
    self->trace = 1;
}

pid$1:::entry, pid$1:::return, fbt:::
/self->trace/
{
    printf("%s", curlwpsinfo->pr_syscall ?
        "K" : "U");
}

pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}
```

Figure 2: Script to follow *all* activity — user-level *and* kernel-level — from a specified user-level function. This script uses the D macro argument variables “\$1” and “\$2” to allow the target process identifier and the user-level function to be specified as arguments to the script.

DTrace also allows for tracing of data from user processes. The `copyin()` and `copyinstr()` subroutines can be used to access data from the current process. For example, the following script aggregates on the name (`arg0`) passed to the `open(2)` system call:

```
syscall::open:entry
{
    @files[copyinstr(arg0)] = count();
}
```

By tracing events in both the kernel and user processes, and combining data from both sources, DTrace provides the complete view of the system required to understand systemic problems that span the user/kernel boundary.

8 Speculative Tracing

In a tracing framework that offers coverage as comprehensive as that of DTrace, the challenge for the user quickly becomes figuring out what *not* to trace. In DTrace, the primary mechanism for filtering out uninteresting events is the predicate mechanism discussed in Section 3.2. Predicates are useful when it is known at the time that a probe fires whether or not the probe event is interesting. For example, if one is only interested in activity associated with a certain process or a certain file descriptor, one can know when the probe fires if it is associated with the process or file descriptor of interest.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
0  -> XEventsQueued                U
0  -> _XEventsQueued                U
0  -> _X11TransBytesReadable        U
0  <- _X11TransBytesReadable        U
0  -> _X11TransSocketBytesReadable  U
0  <- _X11TransSocketBytesReadable  U
0  -> ioctl                        U
0  -> ioctl                        K
0  -> getf                          K
0  -> set_active_fd                 K
0  <- set_active_fd                 K
0  <- getf                          K
0  -> get_uadatamodel               K
0  <- get_uadatamodel               K
...
0  -> releasef                      K
0  -> clear_active_fd               K
0  <- clear_active_fd               K
0  -> cv_broadcast                  K
0  <- cv_broadcast                  K
0  <- releasef                      K
0  <- ioctl                        K
0  <- ioctl                        U
0  <- _XEventsQueued                U
0  <- XEventsQueued                 U
```

Figure 3: Example output of the script from Figure 2, assuming that the script were named “all.d.” Note the crossings of the user/kernel boundary after the first “`ioctl`” and before the last “`ioctl`,” above: while other instrumentation frameworks allow for some unified tracing, this is perhaps the clearest display of control flow across the user/kernel boundary.

However, there are some situations in which one may not know whether or not a given probe event is interesting until some time *after* the probe fires.

For example, if a system call is failing sporadically with a common error code (e.g. `EIO` or `EINVAL`), one may wish to better understand the code path that is leading to the error condition. To capture the code path, one could enable every probe — but only if the failing call can be isolated in such a way that a meaningful predicate can be constructed. If the failures were sporadic or non-deterministic, one would be forced to record all events that *might* be interesting, and later postprocess the data to filter out the ones that were not associated with the failing code path. In this case, even though the number of interesting events may be reasonably small, the number of events that must be recorded is very large — making postprocessing difficult if not impossible.

To address this and similar situations, DTrace has a facil-

```
#pragma D option flowindent

syscall::ioctl:entry
/pid != $pid/
{
    self->spec = speculation();
}

fbt:::
/self->spec/
{
    speculate(self->spec);
    printf("%s: %d", execname, errno);
}

syscall::ioctl:return
/self->spec && errno != 0/
{
    commit(self->spec);
    self->spec = 0;
}

syscall::ioctl:return
/self->spec && errno == 0/
{
    discard(self->spec);
    self->spec = 0;
}
```

Figure 4: A script to speculatively trace all functions called from `ioctl(2)` system calls that return failure. The speculation function returns an identifier for a new speculative tracing buffer; the speculate function indicates that subsequent data-recording expressions in the clause are to be recorded to the specified speculative buffer. This script uses the “\$pid” variable to avoid tracing any failing `ioctl` calls made by `dtrace` itself.

ity for *speculative tracing*. Using this facility, one may tentatively record data; later, one may decide that the recorded data is interesting and *commit* it to the principal buffer, or one may decide that the recorded data is uninteresting, and *discard* it. As an example, Figure 4 contains a script that uses speculative tracing to capture details of only those `ioctl(2)` system calls that return failure; Figure 5 contains example output of this script.

9 Experience

DTrace has been used extensively inside Sun to understand system behavior in both development and production environments. One production environment in which DTrace has been especially useful is a SunRay server in Broomfield, Colorado. The server — which is run by Sun’s IT organization and has 10 CPUs, 32 gigabytes of memory, and approximately 170 SunRay

```
# dtrace -s ./ioctl.d
dtrace: script './ioctl.d' matched 27778 probes
CPU FUNCTION
0 -> ioctl                dhcpagent: 0
0 -> getf                 dhcpagent: 0
0 -> set_active_fd        dhcpagent: 0
0 <- set_active_fd        dhcpagent: 0
0 <- getf                 dhcpagent: 0
0 -> fop_ioctl            dhcpagent: 0
0 -> ufs_ioctl            dhcpagent: 0
0 <- ufs_ioctl            dhcpagent: 0
0 <- fop_ioctl            dhcpagent: 0
0 -> releasef             dhcpagent: 0
0 -> clear_active_fd      dhcpagent: 0
0 <- clear_active_fd      dhcpagent: 0
0 -> cv_broadcast         dhcpagent: 0
0 <- cv_broadcast         dhcpagent: 0
0 <- releasef             dhcpagent: 0
0 -> set_errno            dhcpagent: 0
0 <- set_errno            dhcpagent: 25
0 <- ioctl                dhcpagent: 25
```

Figure 5: Example output from running the script from Figure 4. The output includes the full function trace from *only* the failing calls to `ioctl` — which in this case is an `ioctl` from the DHCP client daemon, `dhcpagent(1M)`, failing with `ENOTTY` (25).

users — was routinely exhibiting sluggish performance. DTrace was used to resolve many performance problems on this production system; the following is the detailed description of the resolution of one such problem.

By looking at the output of `mpstat(1M)`, a traditional Solaris monitoring tool, it was noted that the number of cross-calls per CPU per second was quite high. (A cross-call is a function call directed to be performed by a specified CPU.) This led to the natural question: who (or what) was inducing the cross-calls? Traditionally, there is no way to answer this question concisely. The DTrace “sysinfo” provider, however, is an SDT-derived provider that can dynamically instrument every increment of the counters consumed by `mpstat`. So by using DTrace and `sysinfo`’s “xcalls” probe, this question can be easily answered:

```
sysinfo::xcalls
{
    @[execname] = count();
}
```

Running the above gives a table of application names and the number of cross-calls that each induced; running it on the server in question revealed that virtually all application-induced cross calls were due to the “Xsun” application, the Sun X server. This wasn’t *too* surprising — as there is an X server for each SunRay user, one would expect them to do much of the machine’s work.

Still, the high number of cross-calls merited further investigation: what were the X servers doing to induce the cross-calls? To answer this question, the following script was written:

```
syscall:::entry
/execname == "Xsun"/
{
    self->sys = probefunc;
}

sysinfo:::xcalls
/execname == "Xsun"/
{
    @[self->sys != NULL ?
    self->sys : "<none>"] = count();
}

syscall:::return
/self->sys != NULL/
{
    self->sys = NULL;
}
```

This script uses a thread-local variable to keep track of the current system call name; when the `xcalls` probe fires, it aggregates on the system call that induced the cross-call. In this case, the script revealed that nearly all cross-calls from “Xsun” were being induced by the `munmap(2)` system call. The fact that `munmap` activity induces cross-calls is not surprising (memory demapping induces a cross call as part of TLB invalidation), but the fact that there was so much `munmap` activity (thousands of `munmap` calls per second, system wide) was unexpected.

Given that ongoing `munmap` activity must coexist with ongoing `mmap(2)` activity, the next question was what were the X servers `mmap`’ing? And were there some X servers that were `mmap`’ing more than others? Both of these questions can be answered at once:

```
syscall:::mmap:entry
/execname == "Xsun"/
{
    @[pid, arg4] = count();
}

END
{
    printf("%9s %13s %16s\n",
    "PID", "FD", "COUNT");
    printa("%9d %13d %16@d\n", @);
}
```

This script aggregates on both process identifier and `mmap`’s file descriptor argument to yield a table of process identifiers and `mmap`’ed file descriptors. It uses the special DTrace END probe and the `printa` function to

precisely control the output. Here is the tail of the output from running the above D script on the production SunRay server:

PID	FD	COUNT
26744	4	50
2219	4	56
64907	4	65
23468	4	65
45317	4	68
11077	4	1684
63574	4	1780
8477	4	1826
55758	4	1850
38710	4	1907
9973	4	1948

As labelled above, the first column is the process identifier, the second column is the file descriptor, and the third column is the count. (`dtrace(1M)` always sorts its aggregation output by aggregation value.) The data revealed two things: first, that all of the `mmap` activity for each of the X servers originated from file descriptor 4 in each. And second, that six of the 170 X servers on the machine were responsible for most of the `mmap` activity. Using traditional process-centric tools (e.g., `pf(1)`) revealed that in each X server file descriptor 4 corresponded to the file “/dev/zero,” the `zero(7D)` device present in most UNIX variants. `mmap`’ing /dev/zero is a technique for allocating memory, but why were the X servers allocating (and deallocating) so much memory so frequently? To answer this, we wrote a script to aggregate on the user stack trace of the X servers when they called `mmap`:

```
syscall:::mmap:entry
/execname == "Xsun"/
{
    @[ustack()] = count();
}
```

Running this yields a table of stack traces and counts. In this case, *all* Xsun `mmap` stack traces were identical:

```
libc.so.1'mmap+0xc
libcfb32.so.1'cfb32CreatePixmap+0x74
ddxSUNWsunray.so.1'newt32CreatePixmap+0x20
Xsun'ProcCreatePixmap+0x118
Xsun'Dispatch+0x17c
Xsun'main+0x788
Xsun'_start+0x108
```

The stack trace indicated why the X servers were allocating (and deallocating) memory: they were creating (and destroying) Pixmap. This answered the immediate question, and raised a new one: what applications were ordering their X servers to create and destroy Pixmap? Answering this required a somewhat more sophisticated script:


```

syscall::poll:entry
/execname == "Xsun"/
{
    self->interested = 0;
}

syscall::mmap:entry
/execname == "Xsun"/
{
    self->interested = 1;
}

sched::wakeup
/self->interested/
{
    @[args[1]->pr_fname] = count();
}

```

This script exploits some implementation knowledge of the X server. An X server works by calling `poll(2)` on its connections to wait for requests; when a request arrives, the X server (a single-threaded process) processes the request and sends the response. Sending the response causes the X server to awaken the blocking client, after which the X server again polls on its connections. To determine for whom the X servers were creating Pixmaps, we set a thread-local variable (“interested”) when the X server called `mmap`. We then enabled the “wakeup” probe in the “sched” provider. The sched provider is an SDT-derived provider that makes available probes related to CPU scheduling; the wakeup probe fires when one thread wakes another.⁷ If the X server woke another thread and `interested` was set, we aggregated on the process that we were waking. The core assumption was that the process that the X server awakened immediately after having performed an `mmap` was the process for whom that `mmap` was performed.

Running the above on the production SunRay server produced the following (trimmed) output:

```

***
gedit                25
soffice.bin          26
netscape-bin        44
gnome-terminal       81
dsdm                 487
gnome-smproxy        490
metacity             546
gnome-panel          549
gtik2_applet2        6399

```

This output was the smoking gun — it imme-

⁷In the absence of the sched provider, we would have enabled the FBT probe in the kernel’s routine to awaken another thread, “`sleepq_unlink()`” — but using the well-defined sched provider [12] requires no kernel implementation knowledge.

diately focussed all attention on the application “`gtik2_applet2`,” a stock ticker applet for the GNOME desktop. A further DTrace script that aggregated on user stack revealed the source of the problem: `gtik2_applet2` was creating (and destroying) an X graphics context (GC) *every 10 milliseconds*.⁸ As any X programmer knows, GC’s are expensive server-side objects — they are not to be created with reckless abandon [5]. While there were only six instances of `gtik2_applet2` running on the SunRay server, *each* was inducing this expensive operation from their X servers (and subsequently from the operating system) one hundred times per second; taken together, they were having a substantial effect on system performance. Indeed, stopping the six `gtik2_applet2` processes dramatically improved the system’s performance: cross-calls dropped by 64 percent, involuntary context switches dropped by 35 percent, system time went down 27 percent, user time went down 37 percent and idle time went up by 15 percent.

This was a serious (and in retrospect, glaring) performance problem. But it was practically impossible to debug with traditional tools because it was a *systemic* problem: the `gtik2_applet2` processes were doing very little work themselves — they were inducing work on their behalf from other components of the system. To root-cause the problem, we made extensive use of aggregations and thread-local variables, two features unique to DTrace.

10 Future Work

DTrace provides a stable and extensible foundation for future work to enhance our ability to observe production systems. We are actively developing extensions to DTrace, including:

- Performance counters. Modern microprocessors such as SPARC and x86 export performance counter registers that can be programmed to count branch mispredicts, cache misses, and other processor events. We plan to implement a DTrace provider that exports performance counter information and allows it to be accessed in D from a probe action.
- Helper actions. Complex middleware may wish to assist DTrace with actions that require knowledge specific to the middleware. We have developed a prototype of such a helper action that permits applications to provide assistance for DTrace in obtain-

⁸See http://bugzilla.gnome.org/show_bug.cgi?id=99696 for details.

ing a user-level stack trace. We have implemented the helper action in the Java Virtual Machine, allowing for `ustack` to obtain a user-level stack trace that contains both Java and C/C++ stack frames.

- User lock analysis. The pid provider can instrument any function in a user process, including user-level synchronization facilities. We have developed a prototype user-level equivalent to the kernel `lockstat(1M)` utility, dubbed `plockstat`, that can perform dynamic lock-contention analysis of multi-threaded user processes.

11 Conclusions

We have described DTrace, a new facility for dynamic instrumentation of both user-level and kernel-level software in production systems. We have described the principal features of DTrace, including the details of D, its high-level control language. Although there remain other important features of DTrace for which space did not permit a detailed description (e.g. postmortem tracing, boot-time tracing) we have highlighted the major advances in DTrace over prior work in dynamic instrumentation: thread-local variables, associative arrays, data aggregation, seamlessly unified user/kernel-level tracing, and speculative tracing. We have demonstrated the use of DTrace in root-causing an actual, serious performance problem on a production system — a problem that could not have been root-caused in a production environment prior to this work.

Acknowledgements

Many people at Sun were invaluable in the development of DTrace. We are especially grateful to Bart Smaalders, Gavin Maltby, Jon Haslam, Jonathan Adams, and Bill Moore; their experience, ideas, and tireless advocacy were integral to the success of DTrace. Further, we are grateful to Jarod Jenson of Aeysis, Inc., who agreed to be the Alpha customer for DTrace; it has been singularly rewarding to see Jarod using DTrace to find previously undiagnosable system performance problems. Many people at Sun reviewed drafts of this paper; in particular, it was much improved by the detailed comments of Val Henson, Gavin Maltby, Eric Lowe, Jon Haslam, and Glenn Skinner.

References

- [1] Mikhail Auguston, Clinton Jeffery, and Scott Underwood. A monitoring language for run time and post-mortem behavior analysis and visualization. In *5th International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, 2003.
- [2] Brian Bershad, Stefan Savage, Przemyslaw Parzyk, Emin Gun Sirer, David Becker, Marc Ficuzynski, Craig Chambers, and Su-

- san Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.
- [3] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.
- [4] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Gonçalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, November 1997.
- [5] Eric F. Johnson and Kevin Reichard. *Professional Graphics Programming in the X Window System*. MIS Press, Portland, OR, 1993.
- [6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kirsten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.
- [7] Barton P. Miller, 2003. Personal communication.
- [8] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
- [9] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *Proceedings of the FREENIX Track*, June 2001.
- [10] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.
- [11] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM Symposium on Programming Languages Design and Implementation*, 1994.
- [12] Sun Microsystems, Santa Clara, California. *Solaris Dynamic Tracing Guide*, 2004.
- [13] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.
- [14] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC'2003 Conference CD*, 2003.
- [15] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1999.
- [16] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.
- [17] Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel Degenais. relays: An efficient unified approach for transmitting data from kernel to user space. In *Proceedings of the Ottawa Linux Symposium 2003*, July 2003.

Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging *

Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews and Yuanyuan Zhou
{smsriniv, kandula, crandrws, yyzhou}@cs.uiuc.edu

Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL 61801

ABSTRACT

Software robustness has significant impact on system availability. Unfortunately, finding software bugs is a very challenging task because many bugs are hard to reproduce. While debugging a program, it would be very useful to rollback a crashed program to a previous execution point and deterministically re-execute the “buggy” code region. However, most previous work on rollback and replay support was designed to survive hardware or operating system failures, and is therefore too heavyweight for the fine-grained rollback and replay needed for software debugging.

This paper presents *Flashback*, a lightweight OS extension that provides fine-grained rollback and replay to help debug software. Flashback uses shadow processes to efficiently roll back in-memory state of a process, and logs a process’ interactions with the system to support deterministic replay. Both shadow processes and logging of system calls are implemented in a lightweight fashion specifically designed for the purpose of software debugging.

We have implemented a prototype of Flashback in the Linux operating system. Our experimental results with micro-benchmarks and real applications show that Flashback adds little overhead and can quickly roll back a debugged program to a previous execution point and deterministically replay from that point.

1 Introduction

As rapid advances in computing hardware have led to dramatic improvements in computer performance, issues of reliability, maintainability, and cost of ownership are becoming increasingly important. Unfortunately, software bugs are as frequent as ever, accounting for as much as 40% of computer system failures [45]. Software

bugs may crash a production system, making services unavailable. Moreover, “silent” bugs that run undetected may corrupt valuable information. According to the National Institute of Standards and Technology [48], software bugs cost the U.S. economy an estimated \$59.5 billion annually, approximately 0.6% of the gross domestic product! Given the magnitude of this problem, the development of effective debugging tools is imperative.

Software debugging has been the focus of much research. Popular avenues of such research include detection and analysis of data races [7, 23, 46, 63, 68, 69, 74], static compiler-based techniques to detect potential bugs [20, 24, 31, 36, 64, 76] possibly aided by static checking of user-directed rules [19, 27, 81], run-time checking of data types to detect some classes of memory-related bugs [41, 49], and more extensive run-time checks to detect more complex program errors [28, 51]. These studies have proposed effective solutions to statically or dynamically detect certain types of software bugs.

Even though previous solutions have shown promising results, most software bugs still rely on programmers to interactively debug using tools such as gdb. Interactive debugging can be a very challenging task because some bugs occur only after hours or even days of execution. Some of them occur only with a particular combination of user input and/or hardware configurations. Moreover, some bugs, such as data races, are particularly hard to find because they only occur with a particular interleaved sequence of timing-related events.

These problems motivate the need for low-overhead debugging support that allows programmers to rollback to a previous execution point and re-execute the buggy code region. A deterministic replay recreates the precise conditions that lead to the bug and helps to understand the causes of the bug. In most debugging tools today, if an error occurs, the program needs to be restarted from the very beginning and may take hours or even days to reach the buggy state. If the bug is time-related, the bug may not occur during re-execution. It would be very useful if an interactive debugger such as gdb can periodically checkpoint the process state of the debugged program during its dynamic execution. If an error occurs, the programmer can request gdb to rollback to a previ-

*This work was supported in part by NSF under grants CCR-0325603, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; by an IBM SUR grant; and by additional gifts from IBM and Intel.

ous state and then deterministically replay the program from this state so that the programmer can see how the bug manifests in order to catch its root cause.

Though system support for rollback and replay has been studied in the past, most previous approaches are too heavy-weight to support software debugging. The main reason is that these approaches are geared toward surviving hardware or operating system failures. Therefore, most of these systems checkpoint program state to secondary storage such as disk, remote memory or non-volatile memory [3, 10, 12, 34, 37, 38, 39, 54, 61, 77, 79, 82]. Correspondingly, these systems incur far higher overhead than is necessary or permissible to support software debugging. Unlike hardware/OS failures, we only need to rollback and replay a program when it crashes due to software bugs. Moreover, most previous systems cannot afford frequent checkpointing because of the high overheads involved in these approaches. As a result, applications may have to roll back to a point in the distant past (e.g., 1-2 hours ago).

Besides checkpointing systems, other work on rollback support – such as transaction support for main-memory data structures [11, 29, 40, 43, 60, 62], system recovery [9, 42, 65, 72] or logging and replay of system events [6, 33, 50, 66, 70] – either have problems similar to previous checkpointing systems or require applications to be rollback-aware. These limitations hinder the effectiveness of these solutions for software debugging of general programs.

In this paper, we present a lightweight OS extension called *Flashback* that provides rollback and deterministic replay support for software debugging. In order to efficiently capture the in-memory state of an executing process, Flashback uses *shadow processes* to replicate a program's execution state. Moreover, Flashback also captures the interactions between a program and the rest of the system – such as system calls, signals, and memory mapped regions – to allow for subsequent deterministic re-execution. We have developed a prototype of our proposed solution in the Linux operating system that implements a subset of the features. Our experimental results with micro-benchmarks and real applications show that our system adds little overhead and can quickly roll back to a previous execution point.

As an example of how deterministic replay support can be used for debugging, we also explore the necessary extensions to gdb in order to provide user support for checkpointing, rollback and deterministic replay. These extensions will allow programmers to roll back a program to a previous state when something has gone awry, and interactively replay the buggy code region. With such support, the programmer does not need to restart the execution of the program or to worry about the reproducibility of the bug.

This paper is organized as follows. Section 2 describes the motivation and background of our work. Section 3 presents an overview of Flashback, and sections 4 and 5 describe in greater detail our approach for rollback of

state and deterministic replay. Section 6 presents the experimental results. Section 7 discusses the modifications that have been made to gdb in order to control logging, rollback and recovery from within the debugger. Section 8 concludes the paper with a brief discussion of our experience as well as plans for future work.

2 Background and Related Work

Our work builds upon two groups of research: system support for debugging and system support for rollback. In this section we discuss closely related work done in these two directions.

2.1 System Support for Debugging

Software debugging has been the subject of substantial research and development. Existing approaches mainly include compile-time static checking, run-time dynamic checking and hardware support for debugging. Some representative compile-time static checkers were proposed by Wagner [75, 76], Lujan [44] Evans [21], Engler [19, 27, 81]. Examples of run-time dynamic checkers include Rational's Purify [30], KAI's Assure [35], Lam et. al.'s DIDUCE [28, 51] and several others [41, 49, 15, 53, 58, 63]. Recently, several hardware architecture techniques have been proposed to detect bugs [26, 1, 14, 47, 56].

While these compile-time, run-time or hardware techniques are very useful in catching certain types of bugs, many bugs still cause the programmer to rely on interactive debuggers such as gdb. To characterize timing-related bugs such as race conditions, simply rerunning the program with the same input may not reproduce the same bug. Moreover, some bugs may appear only after running the program for several hours, making the debugging process a formidable task. To understand and find root causes of such bugs, it is very useful to provide system support for reproducing the occurring bug, which may only appear for a particular combination of user inputs and configurations or after a particular interleaved sequence of time-related events.

One effective method to reproduce a bug is to roll back to a previous execution state in the vicinity of the buggy code, and deterministically replay the execution either interactively inside a debugger or automatically with heavy instrumentation. This requires an efficient rollback and deterministic replay mechanism.

2.2 System Support for Rollback

Rollback capability is provided in many systems including checkpointing systems, main-memory transaction systems and software rejuvenation.

Checkpointing has been studied extensively in the past. Checkpointing enables storing the previous exe-

cution state of a system in a failure-independent location. When the system fails, the program can restart from the most recent checkpoint in either a different machine or the same machine after fixing the cause of the failure. Since most checkpointing systems assume that the entire system may fail, checkpoint data is stored either in disks [12, 34, 37, 38, 39, 79, 61], remote memory [3, 54, 82] and non-volatile or persistent memory [10, 80]. As a result, most checkpoint systems incur high overhead and cannot afford to take frequent checkpoints. They are, therefore, too heavy-weight to support rollback for software debugging.

Systems that provide transaction support for main-memory data structures also allow applications to rollback to a previous execution point [11, 29, 43, 60, 62]. For example, Lowell and Chen have developed a system that provides transaction support in the Rio Vista recoverable virtual memory system [11, 43]. Most of these approaches require applications to be written using the transaction programming model; consequently they cannot be conveniently used for debugging a general program.

Borg et al developed a system [5] that provides fault tolerance by maintaining an inactive backup process. In the event of a system failure, the backup process can take over the execution of a process that crashes. The backup process is kept up-to-date by making available to it all the messages that the active process received. Their implementation is based on the assumption that two processes starting from the same initial state will perform identically upon receiving the same input. While this assumption holds for recovery-based systems, it is not the case for general software since the state of the rest of the system may have changed in the meantime. Deterministic replay of a process requires that it receive the same non-deterministic events during replays as during the original run. These events include responses to system calls, shared memory accesses, signals, network messages et al.

Recovery-oriented computing [25, 52] is a recent research initiative that adopts the approach that errors are inevitable, so support for recovering from errors is essential for developing and validating highly available software. Though this is an interesting approach to software availability, most studies in software rejuvenation so far [4, 32, 57] have focused on restarting the whole application rather than fine-grained rollback. Crash-only software [8] is a recent approach to software development that improves the availability of software by using component building blocks that can crash and restart quickly instead of aiming for fault tolerance. These studies focus more on minimizing mean-time-to-recovery (MTTR) than on software debugging.

Feldman and Brown developed a system for program debugging [22] that periodically checkpoints the memory state of a process by keeping track of pages touched by the process. They propose using this system for program restart and comprehensive execution path logging. But their mechanism involves changes to the compiler,

loader, standard library and the kernel. It tracks all memory accesses via code instrumentation and thereby this approach is very heavy-weight. Further, they do not provide deterministic replay; therefore, some errors may not manifest themselves during subsequent re-execution.

Russinovich[59] suggests a lightweight approach to log nondeterministic accesses to shared memory by merely replaying the interleaved order of processes sharing the memory deterministically. An application is instrumented to obtain fine-grained software instruction counters and the OS has to record the location of context switches. This technique can be potentially used by FlashBack to support the replay of shared-memory multi-processed program.

ReVirt[17] is a novel approach to intrusion analysis that encapsulates applications within a virtual OS that itself runs as a process in the guest OS. This technique decreases the size of the trusted computing base (TCB) and allows precise logs to be maintained by the guest OS. Flashback is significantly different from ReVirt. First, debugging support needs to checkpoint application state on timescales(minutes) that are several orders of magnitude smaller than in ReVirt(days). Second, unlike ReVirt which has to contend with malicious intruders by logging "everything", Flashback need only log changes that are made by the application being debugged and external events that affect its operation.

The constraints with existing system support for rollback motivate the need for a new lightweight, fine-grained rollback and deterministic replay solution specifically designed for software debugging.

3 Overview of Flashback

Flashback provides three basic primitives for debugging, *Checkpoint()*, *Discard(x)* and *Replay(x)*.

- *stateHandle = Checkpoint ()*: Upon this call, the system captures the execution state at the current point. A state handle is returned so that the program can later use it for rollback.
- *Discard (stateHandle)*: Upon this call, the captured execution state specified by *stateHandle* is discarded. The program can no longer roll back to this state.
- *Replay (stateHandle)*: Upon this call, the process is rolled back to the previous execution state specified by *stateHandle* and the execution is deterministically replayed until it reaches the point where *Replay()* is called.

To provide the above primitives, Flashback uses *shadow processes* to efficiently capture the in-memory execution state of a process at the specified execution point. The main idea of shadow process is to fork a new process at the specified execution point and this new process maintains the copy of the process's execution state

in main memory. Once a shadow process is created, it is suspended immediately. If rollback is requested, the system kills the current active process and creates a new active process from the shadow process that captured the specified execution state. Since Flashback does not attempt to recover from system crashes or hardware failure, there is no need to store the shadow process onto disk or other persistent storage. This reduces the overhead of the checkpoint process significantly. Moreover, copy-on-write is used to further reduce the overhead.

While our method of checkpointing allows the in-memory state of a process to be reinstated, the process may not see the same set of open file descriptors or network connections during re-execution. Even if the state of file descriptors can be reproduced, it is still a cumbersome task to restore the contents of the file to the original state, and to ensure that network connections will respond exactly as in the original execution. Similarly, during replay it may be undesirable to let the process affect the external environment again by, say, deleting files or modifying their content.

In order to support deterministic replay of a rolled back process, we adopt an approach wherein we record all interactions that the executing process has with the environment. During replay, the logged information is used to ensure that the re-execution appears “identical” to the original run. When a checkpoint is initiated using the `checkpoint` primitive, in addition to capturing in-memory execution state, the system also records the interactions between the process and its environment. During replay, the previously collected information is used to give the process the *impression* that the external environment is responding exactly as it did during the original execution, and that it is affecting the environment in the same way.

Shadow processes can be used in conjunction with the deterministic replay mechanism either within a debugging environment like gdb, or through explicit calls made by the program being debugged:

- *Interactive debugging:* One possible usage scenario is where the debugging platform can periodically capture the state of an executing process by invoking `checkpoint` (similar to the insertion of breakpoints in gdb, for instance). If an error occurs, the programmer can then instruct the debugger to roll back execution to a previously captured state by specifying the time of the earlier checkpoint.
- *Explicit checkpointing and rollback:* An alternate usage scenario is that the programmer takes control of when checkpoints are taken in the code. Figure 1 shows an example of a program where the programmer has inserted explicit invocations to `checkpoint`, `replay` and `discard` primitives.

Automatic checkpoint/rollback support inside an interactive debugger is convenient and requires no changes to the program source code. On the other

```

1  checkpoint(1);
2  fd = open("file.dat", O_WRONLY, 0);
3
4
5  fd = open("file.dat", O_RDONLY, 0);
6
7  n2 = read(fd, buf2, 80);
8  if (n2 > 0)
9      discard(1);
10 else
11     replay(1);

```

Figure 1: Code for a process augmented with primitives

hand, giving the programmer explicit control on checkpoints/rollbacks enables more intelligent and meaningful checkpoint generation.

Figure 1 shows a program in which the programmer calls `checkpoint` in line 1. If the read operation in line 6 fails, the programmer can roll back to the execution state captured at line 1. To help characterize the bug, the execution from line 1 to line 6 can be replayed deterministically by attaching an interactive debugger or switching to a profiling mode with extensive instrumentation. If line 6 succeeds, the checkpoint is discarded.

4 Rollback Using Shadow Processes

4.1 The Main Idea

Flashback creates checkpoints of a process by replicating the in-memory representation of the process in the operating system. This snapshot of a process, known as the *shadow process*, is suspended immediately after creation and is stored within the process structure. A shadow process represents the passive state of the executing process at a previous point, and can be used to unwind the execution of the process by replacing the new execution state with the shadow state and commencing execution in the normal fashion. If a shadow state is not needed anymore, the process can discard it.

The creation of a shadow process for a running process, an event we refer to as *state-capture*, is achieved by creating a new shadow process structure in the kernel, and initializing this structure with the contents of the original process’ structure. The state information captured includes process memory (stack, heap), registers, file descriptor tables, signal handlers and other in-memory state associated with a process. A pointer to this shadow structure is then stored in the original

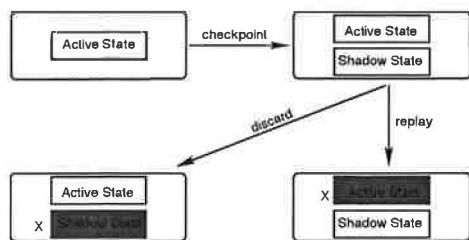


Figure 2: Effect of the primitives on the state of an executing process. When checkpoint is invoked, the process makes a clone of its execution state. Upon discard the shadow is removed; if a rollback occurs, the original execution state is discarded.

process' structure. The new representation of a process with its shadow process is shown in figure 2.

The checkpoint, discard and replay calls are either automatically generated by the debugging infrastructure at specific intervals, or inserted by the programmer in the source code (as shown in the example in the previous section). In case of a discard, the system discards the specified shadow state. If a checkpoint is requested, the system creates a new shadow of the current state and stores it. In the case of a rollback, the process rolls back the execution state to the previously generated shadow process. Figure 2 illustrates the effect of these primitives on the state of a process.

It is possible to maintain two or more shadow processes for an executing process. Multiple shadow processes are useful for progressive rollback and re-execution during debugging [78]. In some cases, when an error occurs, rolling back to only the most recent execution point before replay may not be enough to catch the root of the bug because it could have happened before this execution point. Therefore, it is necessary to roll back further and deterministically restart from an earlier execution point. It is also possible to roll back to the same shadow multiple times and cause additional checkpoints to be taken during replay.

To reduce overhead, shadow process state is maintained using copy-on-write. In other words, state capture proceeds through the creation of an in-memory copy-on-write map of the current state. When a shadow process is created, the virtual memory of the process is marked as read-only. A first write to any page by the active process would trigger the creation of a copy of the old data. This optimization has a couple of benefits. First, the time to create a shadow is significantly reduced by eliminating the need to copy possibly large amounts of memory state. Second, a shadow process occupies little space (in memory). Third, multiple shadows created at different execution points do not need to maintain duplicate copies of the state. Finally, the significant overlap in memory pages between a shadow process and the active process minimizes the impact on the paging behavior of the process due to discard/replay of state. However, writes onto copy-on-write protected memory during execution of the main process does incur overhead. Fortunately, our experimental results pre-

sented in Section 6 show that these overheads are not significant.

4.2 Rolling back multi-threaded processes

Rollback of a multi-threaded process requires special attention. This is because in a multi-threaded environment several components of the process state are implicitly shared across all threads that belong to the same process. For example, threads implemented using the pthread package on Linux, share memory, file descriptors and signal handlers with each other. The only thread-private states are user-space (and kernel) stacks. Such implicit sharing vastly complicates rollback because it is no longer possible for a thread to revert to pristine versions of the shadow state without impacting the execution of other threads.

There are two approaches to support fine-grained rollback of multi-threaded programs. One is to capture the process state for the entire process and roll back all threads to a previous execution point. The second approach is to track thread dependencies such as memory read-write and file read-write dependencies and roll back only those threads that depend on the erroneous thread [2, 16, 18, 67, 70].

Flashback uses the first approach to support rollback of multi-threaded programs. In other words, the underlying system captures the execution state of all threads of a process at a checkpoint. Likewise, when a rollback occurs, Flashback re-instates the execution state of all threads by reverting back to a pristine copy of the shared state. This enables maintenance of consistent state among all threads. Thread synchronization primitives, such as acquiring/releasing locks and semaphore operations are also implicitly rolled back.

Our approach has several advantages over the alternative for software debugging, even though rolling back all the threads of a process when only one of them encounters an error, may seem inefficient. First, our approach is simpler because it does not require complicated logic to keep track of thread dependencies. Tracking thread dependencies is very difficult because concurrent accesses to shared memory are not handled through software or some specialized cache coherence controller. Tracking dependencies requires either hardware support or instrumentation of application binary code to notify the operating system about data sharing. The logic to track dependencies adds overheads to the error-free execution and is also error-prone. Second, to characterize thread synchronization or data races, it might be more informative to roll back all threads and deterministically re-execute all threads step-by-step interactively. Furthermore, the inefficiency of rolling back all threads is encountered only when faults occur - the less common case, while dependency tracking, if done dynamically would lead to overhead on the common case.

4.3 Implementation in Linux

We have modified the *Linux 2.4.22* kernel by adding three new system calls – `checkpoint()`, `discard()` and `replay()` to support rollback and replay. The kernel handles these functions as described earlier. The overhead of these system calls on normal process execution is an important consideration in our implementation.

To capture shadow state, we create a new process control block (`task_struct` in Linux terminology) and initialize it with a copy of the calling process's own structure. This copy involves creation of copy-on-write maps over the entire process memory via the creation of new `mm_struct`, `file_struct` and `signal_struct`. The register contents of the current execution context when it was last in user-space are copied onto the new control block and finally the kernel stack of the new control block is initialized by hand such that the shadow process, when executed, continues execution by returning from the checkpoint system call with a different return value.

The state capture procedure is different from the fork operation in several ways. The primary difference is that after a fork operation, the newly created process is visible to the rest of the system. For instance, the module count is incremented to reflect the fact that the child process is also sharing the same modules. The newly created process is added to the scheduler's run lists and is ready to be scheduled. In contrast, a shadow process is created only for maintaining state. It is not visible to the rest of the system and does not participate in scheduling.

After capturing a shadow state, the calling process returns from the system call and continues execution as normal, with the shadow image in tow. Any changes made to the state after the checkpoint leave the shadow image in its pristine state.

A call to the `discard()` system call deletes a process's shadow image and releases all resources held by it. The `replay()` system call, on the other hand, drops the resources of the current image, and overwrites the process control block with the previously captured shadow image. Since the memory map of the current process changes during the call, the page tables corresponding to the new `mm_struct` are loaded by a call to `switch_mm`.

A subtle result of reinstating the shadow image is that the `replay()` system call never returns to the caller. As soon as the shadow becomes active for the caller, the return address for the `replay()` call is lost (it was part of the speculative state), being replaced instead with the return address of the `checkpoint()` call that corresponds to the state that the process is rolling back to.

When we implemented rollback support for multi-threaded programs in Linux, we encountered many challenges because of the design of Linux thread package that our implementation is based on: `pthread`s. In this thread package, there is a one-to-one mapping between user-space and kernel-space threads, i.e. each user-space thread has an executable process counterpart inside the

kernel. State sharing is achieved by using the `clone` system call to create lightweight processes that share access to memory, file descriptors and signal handlers among other things. POSIX compliance, with respect to delivery of signals (and other requisites), is ensured by creating an LWP thread manager that is the parent of all the threads (LWP's) associated with a process. While the one-to-one mapping allows the thread library to completely ignore the issue of scheduling between threads at user-space, it presents several complications for rollback.

Recall that when one thread attempts to process a checkpoint event, we need to capture the state of all the other threads of that process. Since every user-space thread is mapped to a kernel thread, the other threads may be executing system calls or could be blocked inside the kernel waiting for asynchronous events (sleep-SIGALRM, disk IO etc.). Capturing the transient state of such threads could easily lead to state inconsistency upon rollbacks, such as rolling back to a *sleeping* state when the corresponding kernel timer has already expired¹. It is difficult to capture the state of an execution context from within a different execution context.

We are currently exploring a solution to this problem by explicitly identifying such troublesome scenarios and manipulating the user-space and kernel stacks to ensure that the interrupted system call is re-executed upon rollback. Specifically, threads that are blocked in system calls are checkpointed *as if* they are about to begin execution of this interrupted system call.

Notice that apparently simple solutions that circumvent this problem such as using inter-process communication or explicit barrier synchronization prior to state capture are not applicable. In the former case, IPC mechanisms such as signals and pipes increase the latency of the state capture event because their processing is usually deferred, and is often not deterministic. Barrier synchronization on the other hand, would cause the processing of a state capture event to be delayed until the event is generated on all the threads of a process, which might be unrealistic in certain applications.

5 Replay Using Record-and-Sandbox

5.1 The Main Idea

In order to deterministically replay the execution of a process from a previous execution state, we need to ensure during re-execution that the process perceives no difference in its interaction with the environment. For instance, if the process did a `read` on a file and received a particular array of bytes, during replay, the process should receive the same array of bytes and return value as before, though the file's contents may have already been changed.

¹sleep on Linux is implemented using `nanosleep` which swaps out the process after adding a timer onto the kernel's timer list

Flashback does not ensure exactly the same execution during replay as during the original run. Instead, Flashback provides only an impression to the debugged process that the execution and interaction with the environment appears identical to those during the original run. It is difficult to provide the exact same execution because the external environment, such as network connections or device states, etc, is beyond the control of the operating system. As long as Flashback interacts with the debugged process in the same way, with very high probability, the bug can be reproduced during replay.

A process in Flashback can operate in one of two modes - *log* and *replay*. In the *log* mode the system logs all interactions of the process with the environment. These interactions can happen through system call invocations, memory-mapping, shared memory in multi-threaded processes, and signals. The process enters the log mode when the *checkpoint* primitive is invoked. In the replay mode, the kernel handles system interactions of the process by calling functions that simulate the effect of the original system call invocation. The replay mode is selected when the *replay* primitive is invoked. In this mode, Flashback ensures the interaction between the replayed process and the OS is the same as was logged during the original run.

5.2 System calls

Logging and replay are different for different types of system calls:

- Filesystem-related – Calls such as open, close, read, write, seek
- Virtual memory-related, such as memory allocation, mmap etc.
- Network-related – such as socket creation, polling, send, recv etc.
- Process control – such as exec, fork, exit, wait
- Interprocess communication-related – such as creation and manipulation of message queues and named pipes
- Utility functions – such as getting the time of day

When simulating the effect of a system call, Flashback has to ensure that the values returned by the system call are identical to those returned during the original execution. In addition, the original system call may return some “hidden” values by modifying memory regions pointed to by pointer arguments. For example, the *read()* system call loads the data from the file system into the buffer specified by one of the parameters. These side effects also need to be captured by Flashback. A faithful replay of a system call thus requires Flashback to log all return values as well as side-effects. While somewhat tedious because of the special attention

required by each system call to handling its specific arguments, this support can easily be provided for a large body of system calls.

In Flashback, we intercept system calls invoked by a process during its execution. In order to do this, we replace the default handler for each system call with a function that does the actual logging and replay as shown in figure 3. In logging mode, the function invokes the original call and then logs the return values as well as the side-effects. In replay mode, the function checks to confirm that the same call is being made again, and then makes the same side-effects and returns the logged return value.

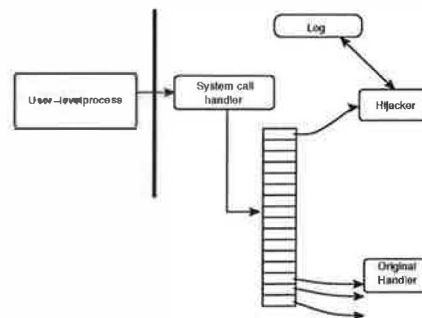


Figure 3: Hijacking System Calls for Logging and Replay in Flashback

A notable exception to bypassing the actual system calls during replay is for calls related to memory management, such as memory mapping and heap management. In this situation we cannot fake memory allocation – if the process accesses a memory location that we have faked the allocation of, then it will result in a segmentation fault. This problem arises because while memory is allocated and deallocated using the *brk()* system call, it may be accessed through direct variable assignments. The changes made to memory locations do not make any permanent changes to the system; i.e. the state is captured by a process’ checkpoint exclusively. As we discuss shortly, however, this may not be the case for files that have been mapped into memory.

Once system calls have been handled, much of the process’ original execution can be replayed. For instance, the process being replayed can read data from files as it did before even though these files may actually have been modified or may not even exist in the system anymore. Similarly, it will receive network packets as it originally did from remote machines. As far as the process is concerned, it believes that these events are happening as they did before in terms of both actual data exchanged and the relative timing of asynchronous events.

5.3 Memory-Mapped Files and Shared Memory

Linux supports two different flavors of shared memory for interprocess communication – System V IPC and BSD mmap. These implementations allow processes to share a single chunk of memory by mapping the shared memory onto their respective memory spaces. BSD mmap allows processes to map a previously opened file into a region of its memory, after which it can access the file using simple memory assignment instructions. When a shared segment is requested, the kernel forces the memory management unit (MMU) to generate a page fault every time a previously unused section of this memory region is accessed. In response to the page fault, the kernel loads one page of data from the file and reads it into the process' memory.

A file may be mapped as either *private* or *shared*. Any changes made to privately mapped files are visible only to that process and do not result in changes to the file. On the other hand, files that are mapped as shared may be modified when the process writes to the memory area. Further, for shared files, changes made to the file by a processes will be immediately visible to other processes that have mapped the same region of the file. Providing replay for shared memory poses problems as a process can access shared memory without making any system calls, making it harder to track changes to the shared memory and fake them later.

One simple solution for handling memory-mapped files is to make copies of pages that are returned upon the first page fault to a memory region mapped to a file. During replay of requests to create memory maps, the memory areas are mapped to dummy files, and page faults are handled by returning saved copies of pages. Due to the lazy demand-paging approach used by Linux, only those pages that are accessed during execution need to be copied, thus drastically reducing the overhead. This approach will not work when the same region of the file is mapped as a shared region by multiple processes, each of which make changes to the region. This approach works for files that have been mapped as private, as well as shared mappings where all changes to the file are made by the process being debugged.

Handling shared file-mappings with multiple processes writing to the file is a more complicated problem, and requires the kernel to force a page fault for every access to the shared region by the process being replayed instead of just the first access as in the earlier case. A possible enhancement to the logging solution would be to set the access rights of a given page to the last process to access it, and thus only fault when another process has accessed the page since this one. This way, several successive reads or updates will only suffer one costly exception instead of many. During replay, however, it would still be required to fault for each access since the other processes might not be around any more to make their changes.

In Flashback, currently, we have implemented the simple solution described earlier. In spite of the enhancement proposed for shared file-maps with multiple writers, we believe that an efficient solution to address this challenge will require support from the underlying architecture. Shared memory can be dealt with using similar mechanisms.

5.4 Multithreaded applications

While the techniques outlined above work for applications with a single thread of control, replaying multithreaded applications poses additional challenges. Logging changes made by a multithreaded application involves logging the changes of each thread of the debugged process. During replay, the interleaving of shared memory accesses and events has to be consistent with the original sequence.

Ensuring that the multiple threads are scheduled in the same relative order during replay is another issue. For multi-threaded applications running on a single processor system, we propose adopting the approach described in [13] for deterministic replay. The basic idea is to record information about the scheduling of the threads during the original execution and use this information during replay to force the same interleaving between thread executions. Since this implementation would also be in the kernel, the physical thread schedule is transparent and can be used in lieu of the logical thread schedule information proposed by [13]. We will implement this in the future in the tool, possibly with the support of architecture-level mechanisms such as those described in [55].

5.5 Signals

Signals are used to notify a process about a specific event, or to force the process to execute a special handling code when an event is detected during its execution. Signals may be sent to a process either by another process or by the kernel itself. Signals are asynchronous and are delivered proactively to a process by the kernel. They may be delivered at any time to a process. Signals present a challenge for deterministic replay because signals are asynchronous events that affect the execution of a process. The replaying mechanism has to ensure that signals are delivered at exactly the same points during re-execution as in the original execution.

Deterministic reproduction of signals may be handled using the approach proposed by Slye and Elnozahy [66], though Flashback does not currently support signal replay. The mechanism outlined in their work makes use of an instruction counter to record the time between asynchronous events. The instruction counter is included in most modern processor systems today. When a signal occurs, the system creates a log entry for it, which includes the value of the instruction counter since the last system call invocation. During replay, Flashback checks

to see if the next log entry corresponds to a signal. If so, then it initializes the instruction counter with the time from the current system call till the signal. When a trap is generated because of timeout, the kernel delivers the signal to the process.

5.6 Implementation in Linux

We have implemented a prototype of Flashback's replay mechanism in Linux-2.4.22. The prototype handles replay of system calls as well as memory-mapped files to a limited extent. In Linux, a user-space process invokes a system call by loading the system call number into the `eax` register and optional arguments in other registers, and then raising a programmed exception with vector 128. The handler for this exception, the system call handler, does several checks and then runs the function indexed in the `sys_call_table` array by the *system call number*. It finally returns the results got from this action to the user process.

We used *syscalltrack* [71], an open-source tool that allows system calls to be intercepted for various purposes such as logging and blocking. The core of the tool has been implemented in a kernel module which "hijacks" the system call table by replacing the default handlers for some system calls with special functions. System call invocations can be filtered based on several criteria such as the process id of the invoking process as well as values for specific arguments. System calls that need to be logged are handled in a number of ways. At one extreme, the special function may log the invocation of the system call and let the call go through to the original handler, while at the other it may block the system call invocation and return a failure to the user process. The actual behavior of the special function is controlled using rules that may be loaded into the kernel.

In our implementation, we added a new action type that the special function can perform, namely the `AT_REPLAY` action for replaying. This action verifies that the system call invocation matches a call that the process originally made, then sets the return value according to the logged invocation and also makes the same side effects on the arguments as before. By doing this, it bypasses the actual system call handler for some system calls and overrides its behavior with that of the simulating function. For other system calls such as the `brk` call, Flashback allows the system calls to be handled by the original system call handler since memory allocations need to be made even during replay.

6 Evaluation

We evaluate our prototype implementation of Flashback using microbenchmarks as well as real applications. The timing data we present were obtained on a 1.8GHz Pentium IV machine with 512KB of L2 cache and 512MB of RAM.

6.1 Overhead of State-Capture

To perform a very basic performance evaluation of the rollback capabilities, we instrumented the `checkpoint()`, `discard()` and `replay()` system calls. We then ran a small program that repeatedly invokes `checkpoint()`, does some simple updates and then either discards the checkpoint by calling `discard()` or rolling back by calling `replay()`.

Figure 4(a) presents the time for the three basic operations: checkpoint, discard and replay. A checkpoint takes around 25-1600 μ s as the amount of state updates between two consecutive checkpoints varied from 4KB to 400MB. Since creation of a shadow process involves creation of a copy-on-write map, the cost is proportional to the size of the memory occupied by the process. Similarly, the cost to discard or replay a shadow is proportional to the size of memory modified by the process.

The costs of discard (replay) are also directly proportional to the number of pages in the corresponding checkpointed state (the current state). This is because both discard and replay involve deletion of one copy-on-write map. Our results show that discard and replay take around 28-2800 μ s when the entire memory is read, and between 28-7500 μ s when the entire data memory is written. The higher costs in the latter case are because the kernel has to return a large number of page frames to its free memory list when the shadow state is dropped/reinstated. Typical applications will of course not modify all pages in their address space between checkpoints, and so the costs of the discard and replay operations will be closer to the lower end of the range shown in Figure 4(b).

An important objective of our rollback infrastructure is to have minimal impact on normal application performance. We therefore consider the data for `checkpoint()` and `discard()` more important than that for `replay()`. This is because the latter is invoked only when errors occur, and will therefore not be part of common-case behavior. Regardless, the overhead imposed by the rollback call is as low as that for shadow state release. This is promising since it indicates we can restore execution state as fast as common case checkpoint discard.

6.2 Overhead due to Logging

In order to evaluate the logging overhead, we wrote a simple test program that employs two threads in order to isolate the impact of the logging overhead. In the program, the parent thread forks and creates a child. It then loads the rules for logging into the framework and notifies the child to begin invoking system calls. The rules allow the kernel to filter system call invocations based on the process ID of the child.

While logging system calls that have side effects on memory regions, such as `read`, `stat` and `getsockopt`, Flashback also needs to record the contents of the buffer

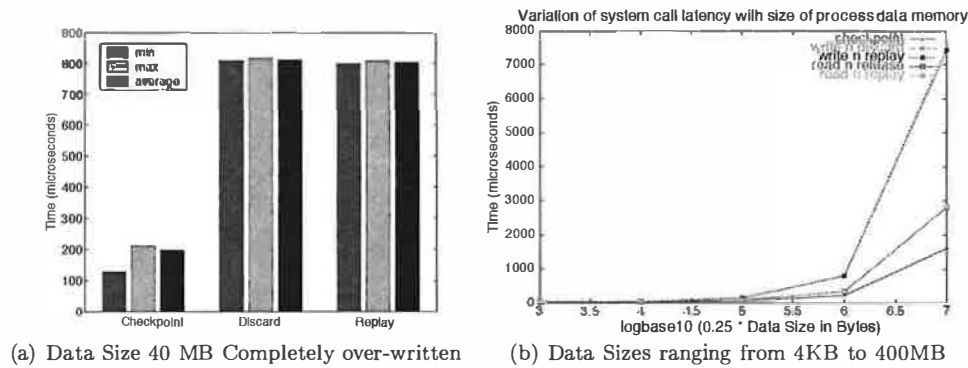


Figure 4: Microbenchmark Results for Shadow Process Creation at different sizes of process data memory

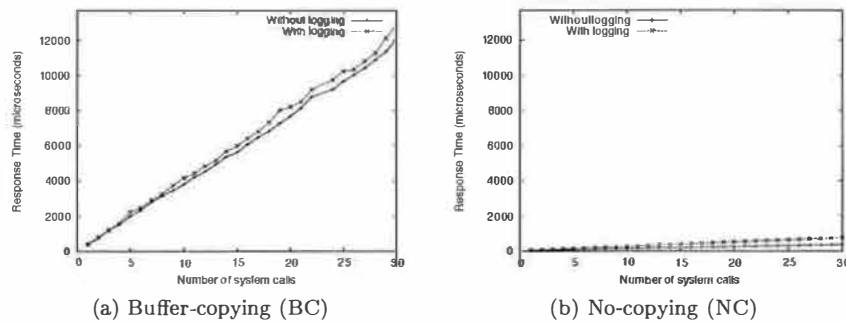


Figure 5: Response time overhead (microseconds) for varying number of system call invocations

or structure. Thus, with regard to logging overhead, there are two groups of system calls, those that cause side effects on some memory regions, and those that simply return a value after performing the intended action. We refer to the first class of system calls as buffer-copying (BC) and the second group as no-copying (NC). For NC system calls, there is no need to record the contents of buffers; just the system call ID and the return values will suffice.

To study the overhead on every system call due to hijacking and logging, we invoked the read and write system calls several times, gradually increasing the number of invocations. In each invocation, the number of bytes read or written is 4 KB. For each run, we start with a clean file cache in order to make the effect of caching on system call overheads consistent. Figures 5 shows the overhead imposed by the sandbox mechanism. The overhead due to sandboxing occurs because of the extra indirection of system calls imposed by Flashback. Instead of being handled directly by the system call handlers, system call requests need to pass through filters and the logging mechanism. The increase in overhead is linear with the number of system calls for both the system calls. The difference in slope between the two lines on the graphs represents the extra per-system-call overhead imposed due to logging. This is around 30 microseconds on an average.

To evaluate the effect that the copying of buffers has on the logging overheads, we invoked the read and write

system calls repeatedly, gradually increasing the number of bytes read or written from 4 KB to 2 MB. The actual number of system calls is small in this case. Figure 6 shows the overhead while varying the amount of data read or written. The overhead for BC and NC system calls is comparable, and the extra copying of buffers does not appear to impose any extra overhead. This is because the contents of the log are buffered, and written to disk asynchronously. In these experiments, the disk cache was warmed since all the data for the files was prefetched before the actual execution. The values therefore reflect reads and writes entirely involving the cache only.

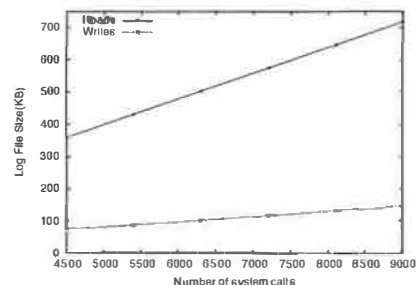


Figure 7: Size of the log file (KB) for varying number of system call invocations

Figure 7 shows the space overhead because of logging

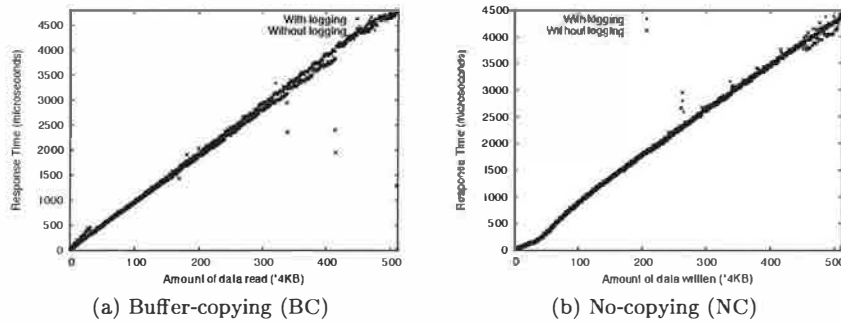


Figure 6: Response time overhead (microseconds) for varying sizes of memory logging

BC and NC calls. As expected, the growth in the size of the log file is linear in terms of the number of system calls, though the slope is greater for BC since more data is written each time.

6.3 Application Results

In order to test our implementation of state-capture in a realistic environment, we measure the performance with the well-known *Apache* web-server. We evaluate the system overhead for both multiprocess version and multithread version of *Apache*. Our evaluation serves to demonstrate two things: first, that fine-grained rollback support is possible, and can be applied to real applications; and second, that the performance impact on common-case execution is minimal.

In all the experiments reported herein, the web server is bottle-necked by the network and is serving data at full network throughput of 100Mbps. We use these experiments to show that off-the-shelf machines (1.8MHz, 512MB RAM) have enough spare cpu cycles to provide fine-grained rollback without affecting client's perceived performance. The server is checkpointed multiple times (typically thrice) during the processing of each request. We essentially create a checkpoint just before reading the HTTP request off a newly accepted socket, before processing a valid HTTP request from an existing connection and before writing out the HTTP response onto the socket. Thus, at any point of time, Flashback maintains as many shadow images as the total number of requests being processed by the server. All data points in this section have been averaged over three runs.

The *Apache* server can be configured to run in a multiprocess or multithread mode. In the former, *Apache* maintains a pool of worker processes to service requests. Each worker process is a single thread and the number of workers in the pool is adapted dynamically based on load estimates. However, in the latter, *Apache* uses a much smaller pool of worker processes, with each worker process consisting of multiple threads implemented by the *pthread* package. We present here performance figures for both configurations of the *Apache* server. In this experiment, the web server checkpoints its state upon the arrival of request for a page, processes the request,

and discards the checkpoint. These results reflect the overhead of capturing state. Since Flashback currently does not support replay of multithreaded execution and shared memory, we disabled logging for replay during these experiments.

To exercise the web server, we use an http request-generating client application, *WebStone* [73], which sends back-to-back requests to a single web server. Each request constitutes a fetch of a single file, randomly selected from a pre-defined "working set". The working set comprised files of sizes varying between 5KB and 5MB, but the majority of requests constituted a fetch of 5KB. The request generating application forks a pre-defined number of client processes, each of which submits a series of random requests to the web server. The server was run on a off-the-shelf 1.8GHz Pentium IV machine, connected to the client via a 100Mbps LAN. Performance was measured in terms of throughput, aggregate response time and load on the server CPU. In all the experiments reported here, the server was operating at the full network throughput of 100Mbps.

We compare the *Apache* web-server on the prototype system with a baseline system running the original version of Linux. Figure 8 shows throughput and response time in Flashback and the baseline system with *Apache* running in multiprocess mode. It is clear from the graphs that there is no significant difference between the client-perceived throughput and response time. When the number of clients is small, Flashback has 10% lower throughput, even though the average response time is the same as the baseline system. However, when the number of clients increases, the difference between baseline and Flashback disappears. In some cases, Flashback performs even better than the baseline system. We consider these small differences well within expected experimental variance, and conclude that the impact of rollback support on *Apache* performance is negligible.

Figure 9 shows the results for the multithread version of *Apache*. As expected, the overheads imposed by Flashback on multithreaded execution are slightly lower than those for the multiprocess version, evidenced by the throughput figures which more closely match one another in most cases. This lower overhead is a direct result of fewer effective system calls, because when one

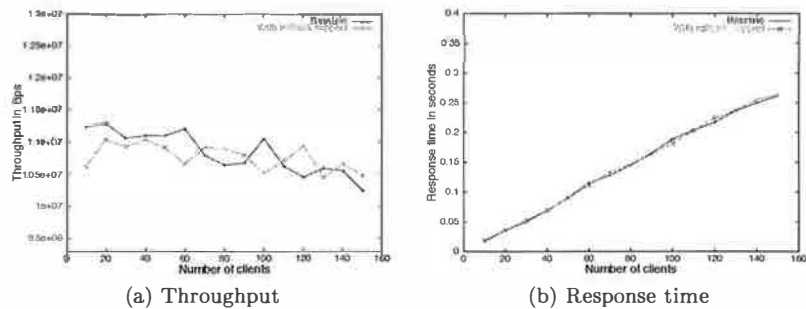


Figure 8: Throughput and response time with Multiprocess Apache web-server. *Baseline* corresponds to the case running in the default Linux system without rollback support, and *With rollback support* corresponds to the Linux kernel modified to include rollback support. The results shown in these figures indicate that throughput and response time are not affected by Flashback. These times reflect state-capture overhead

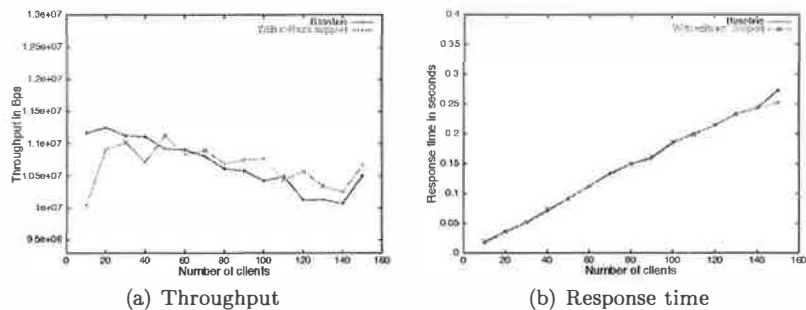


Figure 9: Throughput and response time with multithread Apache web-server. The results shown in these figures indicate that throughput and response time are not affected by Flashback. These times reflect state-capture overhead

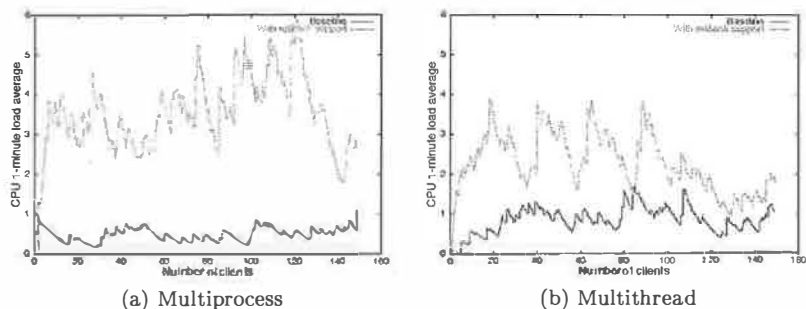


Figure 10: One-minute CPU load averages for the host on which the Apache web-server is running. The curves demonstrate the extra work being performed by the kernel when checkpointing is enabled.

thread undergoes a capture event, the state of all the other threads is automatically captured. Subsequent capture-events on the other threads of this process are treated as *nops* during the lifetime of this shadow process. Hence the number of capture events necessary are much fewer.

Although client-perceived system performance remains almost unaffected, the kernel does perform extra work each time a checkpoint is initiated. Of course, this does not come for free. To quantify the cost, we monitored CPU load average on the machine hosting the web-server. The metric we use measures the average number of processes waiting on the run queue over the last minute, which is an estimate of system load as it statis-

tically captures the amount of time each process spends on the run queue. Figure 10 plots these results for the multiprocess and multithread versions. The graphs expose the overhead in capturing shadow state, which in our evaluation occurs very frequently (once every request received by the server). Note that even though the cpu utilization of the server increases by 2-4 times, the client perceived performance, in both data bytes delivered and time to respond, remains unchanged. We assert that the experimental setup is realistic as modern web-servers are often constrained by network bandwidth and have spare cpu cycles.

In both the multiprocess and multithread configurations, CPU load increases significantly. In the single-

threaded case, the extra load is quite high. This is because a multiprocess Apache webserver uses a collection of separate Unix processes to handle web requests, each of which now captures shadow state when handling a request. In the multithreaded version, the state-capture event occurs once for all threads of execution, because we capture the state of all threads, *en masse*, each time a checkpoint is taken. The smaller number of system calls, and the smaller size of the state captured (per worker thread), together contribute to the multithread configuration exhibiting better CPU load than the multiprocess configuration.

7 Using Flashback in gdb

Using Flashback, it is fairly straightforward to incorporate support for checkpointing, rollback and deterministic replay into a debugging utility such as *gdb*.

We have modified *gdb* to support three new commands – checkpoint, rollback, and discard, for creating checkpoints, to support rollback and deterministic replay of debugged programs. Programmers can set up breakpoints at places where they might want to create checkpoints. At these breakpoints, after seeing the state of the program, they can choose to create a new checkpoint by using the checkpoint command. They can also discard earlier checkpoints, thereby freeing system resources associated with those checkpoints by using the discard command. If they find the system state to be inconsistent, they can roll back to an earlier checkpoint by using the rollback command.

Using Flashback, *gdb* can be made to automatically take periodic checkpoints of the state of the process being executing. New commands are added into the debugger user interface to allow programmers to enable or disable automatic checkpointing during execution of the debugged program. Programmers also have control over the frequency of checkpointing. This frees the programmer from having to insert breakpoints at appropriate locations in the code and explicitly taking checkpoints.

In order to incorporate checkpoints into *gdb*, we made changes to the *target system handling component* and the *user interface* components. The target system handling component handles the basic operations dealing with the actual execution control of the program, stack frame analysis and physical target manipulation. This component handles software breakpoint requests by replacing a program execution with a trap. During execution, the trap causes an exception which gives control to *gdb*. The user can choose to take a checkpoint at this time. *gdb* does this by making a checkpoint system call passing the process ID of the process being debugged. Similarly, for rollback and replay, *gdb* uses the *rollback* and *replay* system calls respectively.

For automatic checkpointing, in addition to these changes, *gdb* maintains a timer that keeps track of time since the last checkpoint. The timeout for the timer can be set by the user. When a timeout occurs, *gdb* check-

points the process.

8 Conclusions and Future Work

In this paper we presented a lightweight OS extension called Flashback to support fine-grained rollback and deterministic replay for the purpose of software debugging. Flashback uses shadow process to efficiently capture in-memory states of a process at different execution points. To support deterministic replay, Flashback logs all interactions of the debugged program with the execution environment. Results from our prototype implementation on real systems show that our approach has small overheads and can roll back programs quickly.

Besides software debugging, our system can also be used to improve software availability by progressively rolling back and re-executing to avoid transient errors [78]. In addition, our approach can be extended to provide lightweight transaction models that require only atomicity but not persistence.

We are in the process of combining Flashback with hardware architecture support for rollback and deterministic replay [56] to further reduce overhead. We are also evaluating Flashback with more applications. Flashback currently only works for programs that run on a single machine. We are exploring ways to extend it to support distributed client-server applications by combining with techniques surveyed by Elnozahy et al. [18].

Flashback including the patches to both Linux and *gdb* will be released to the open source community so that other researchers/developers can take advantage of *Flashback* in interactive debugging.

9 Acknowledgments

We would like to thank Dr Srinivasan Seshan, the shepherd for the paper, for useful suggestions and comments. We also thank the anonymous reviewers for useful feedback, and the Opera group for useful discussions, and Jagadeesan Sundaresan, Pierre Salverda and Arijit Ghosh for their contribution to the project.

REFERENCES

- [1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 234–243, 1991.
- [2] Alvisi and Marzullo. Trade-offs in implementing causal message logging protocols. In *PODC: 15th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1996.
- [3] C. Amza, A. Cox, and W. Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. Proc. of the International Conference on Dependable Systems and Networks., 2000.
- [4] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *IEEE International Computer Performance and Dependability Symposium*, 1998.

- [5] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, volume 17, pages 90–99, 1983.
- [6] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb. 1989.
- [7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
- [8] G. Candea and A. Fox. Crashonly software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.
- [9] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *OSDI*, 2000.
- [10] P. M. Chen, D. E. Lowell, and G. W. Dunlap. Discount checking: Transparent, low-overhead recovery for general applications. Technical report, University of Michigan, Department of Electrical Engineering and Computer Science, July 1998.
- [11] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating systems crashes. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, Massachusetts, 1–5 Oct. 1996. ACM Press.
- [12] Y. Chen, J. S. Plank, and K. Li. Clip: a checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–11. ACM Press, 1997.
- [13] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.
- [14] J.-D. Choi and S. L. Min. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 145–154, 1991.
- [15] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope Parallel Programming Environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.
- [16] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *International Conference on Distributed Computing Systems*, pages 108–115, 1996.
- [17] G. W. Dunlap, S. T. Kind, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 35(SI):211–224, 2002.
- [18] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
- [19] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.
- [20] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, December 1994.
- [21] D. Evans and D. Laroche. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.
- [22] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, Jan. 1989.
- [23] C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.
- [24] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.
- [25] G. Candea et. al. Reducing recovery time in a small recursively restartable system. In *DSN*, 2002.
- [26] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, 1991.
- [27] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI)*, 2002.
- [28] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 2002 Int. Conf. Software Engineering*, pages 291–301, Orlando, FL, May 2002.
- [29] R. Haskin, Y. Malachi, and G. Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems (TOCS)*, 6(1):82–108, 1988.
- [30] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *the Winter USENIX*, 1992.
- [31] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.
- [32] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: analysis, module and applications. In *FTCS-25*, 1995.
- [33] Y. Huang and Y. Wang. Why optimistic message logging has not been used in telecommunication systems. In *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 459–463, June 1995.
- [34] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, Aug. 1988.
- [35] KAI-Intel Corporation. Assure. URL: <http://developer.intel.com/software/products/assure/>.
- [36] S. Kumar and K. Li. Using model checking to debug network interface firmware. In *the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
- [37] K. Li, J. Naughton, and J. Plank. Concurrent real-time checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Seattle, Washington, Mar. 1990.
- [38] K. Li, J. Naughton, and J. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Programming*, 20(3):159–180, June 1991.
- [39] K. Li, J. Naughton, and J. Plank. Low-latency concurrent checkpoint for parallel programs. *IEEE Transactions on Parallel and Distributed Computing*, 1994.
- [40] B. Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, March 1988.
- [41] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, pages 217–232, 2001.
- [42] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and limits of generic recovery. In *OSDI*, 2000.
- [43] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 92–101, New York, Oct.5–8 1997. ACM Press.
- [44] M. Luján, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of Java array bounds checks in the presence of indirection. In *Proceedings of the Joint ACM Java Grande-Iscope Conference*, pages 76–85, 2002.

- [45] E. Marcus and H. Stern. Blueprints for high availability. John Wiley and Sons, 2000.
- [46] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Apr. 1991.
- [47] S. L. Min and J.-D. Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, 1991.
- [48] National Institute of Standards and Technology (NIST), Department of Commerce. Software errors cost u.s. economy \$59.5 billion annually. NIST News Release 2002-10, 2002.
- [49] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.
- [50] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD*, 1993.
- [51] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads, October 2002.
- [52] D. A. Patterson and et. al. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. UC Berkeley CS Tech. Report, UCB//CSD-02-1175, 2002.
- [53] D. Perkovic and P. J. Keleher. A Protocol-Centric Approach to on-the-Fly Race Detection. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1058–1072, 2000.
- [54] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–??, 1998.
- [55] M. Prvulovic and J. Torrellas. Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual Symposium on Computer Architecture*, 2003.
- [56] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation to Debug Software; An Application to Data Races in Multithreaded Codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.
- [57] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 15–28, Banff, Canada, Oct. 2001.
- [58] M. Ronsse and K. D. Bosschere. RecPlay: a Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.
- [59] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 258–266, Jerusalem, Israel, 1996. ACM Press.
- [60] Y. Saito and B. Bershad. A transactional memory service in an extensible operating system. In *USENIX Annual Technical Conference*, 1998.
- [61] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. Technical Report CS-TR-126-87, Department of Computer Science, Princeton University, 1987.
- [62] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, and J. Kistler. Lightweight recoverable virtual memory. In *SOSP*, 1993.
- [63] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [64] E. Schonberg. On-the-fly detection of access anomalies. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*, June 1989.
- [65] M. Seltzer, Y. Endo, and C. Small. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, 1996.
- [66] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pages 250–261, Washington, June 25–27 1996. IEEE.
- [67] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 361–371, Pasadena, California, 1995.
- [68] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.
- [69] J. M. Stone. Debugging concurrent processes: A case study. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1988.
- [70] R. E. Strom and S. A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug. 1985.
- [71] syscalltrack software home page at <http://syscalltrack.sourceforge.net/how.html>.
- [72] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *ASPLOS*, 1994.
- [73] G. Trent and M. Sake. Webstone: The first generation in http server benchmarking. Feb 1995.
- [74] C. v. Praun and T. Gross. Object race detection. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, FL, October 2001.
- [75] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.
- [76] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.
- [77] Y. Wang, P. Y. Chung, Y. Huang, and E. N. Elnozahy. Integrating checkpointing with transaction processing. In *FTCS*, 1997.
- [78] Y. Wang, Y. Huang, W. K. Fuchs, C. Kintala, and G. Suri. Progressive retry for software failure recovery in message-passing applications. *IEEE Transactions on Computers*, 46(10):1137–1141, Oct 1997.
- [79] Y. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *FTCS-25*, 1995.
- [80] M. Wu and W. Zwaenepoel. eNvy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, San Jose, California, Oct. 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.
- [81] Y. Xie and D. Engler. Using redundancies to find errors. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60, 2002.
- [82] Y. Zhou, P. M. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. In *the 13th ACM International Conference on Supercomputing*, June 1999.

Email Prioritization: reducing delays on legitimate mail caused by junk mail

Dan Twining, Matthew M. Williamson, Miranda J. F. Mowbray, Maher Rahmouni
HP Labs, Filton Road, Stoke Gifford, Bristol, BS34 8QZ, UK

dan@dantwining.com, matthew_williamson@alum.mit.edu, {miranda.mowbray, maher.rahmouni}@hp.com

April 26, 2004

Abstract

In recent years the volume of junk email (spam, virus etc.) has increased dramatically. These unwanted messages clutter up users' mailboxes, consume server resources, and cause delays to the delivery of mail. This paper presents an approach that ensures that non-junk mail is delivered without excessive delay, at the expense of delaying junk mail.

Using data from two Internet-facing mail servers, we show how it is possible to simply and accurately predict whether the next message sent from a particular server will be good or junk, by monitoring the types of messages previously sent. The prediction can be used to delay acceptance of junk mail, and prioritize good mail through the mail server, ensuring that loading is reduced and delays are low, even if the server is overloaded.

The paper includes a review of server-based anti-spam techniques, and an evaluation of these against the data. We develop and calibrate a model of mail server performance, and use it to predict the performance of the prioritization scheme. We also describe an implementation on a standard mail server.

1 Introduction

In recent years the volume of junk mail (spam, virus, etc.) has increased dramatically. According to MessageLabs, the volume of spam increased by 77% in 2003, and virus-carrying emails increased by 84% [18]. This increase in volume taxes the resources of servers responsible for routing mail, and increases the transit time of emails. During virus/worm attacks these delays can increase to unacceptable levels [2]. This paper presents

an approach that ensures that most of the “good” mail is processed quickly, at the expense of larger delays for junk mail.

Unfortunately, this problem is only going to get worse. The volume of email traffic increases over time [34], but the volume of spam increases much faster. MessageLabs predict that in 2004 70% of all messages will be spam, as compared to (only) 55% as of November 2003 [17]. There are other reasons why it is getting worse. First is the convergence of viruses and spam, where virus-infected machines are being sold and used as spam relays [37]. Secondly, sites that offer blacklists (lists of IP addresses used by spammers) have been subject to denial of service attacks [28], some launched from viruses [30]. Thirdly, there is a continual arms race between spam and content filtering programs, with filtering always behind and missing some spam [20, 11].

Our approach is to enable good mail to reach the desktop without delay, in contrast to most existing approaches to the problem of spam, which concentrate on preventing junk mail from reaching the desktop. Some of the prevention mechanisms reduce server load (and so reduce delays), for example the use of blacklists and other heuristics to refuse connections from spamming servers. Others increase server load, for example content checking and filtering at the server. As users are intolerant of any mistakes in classification (particularly false positives—legitimate mail classed as spam), and as spammers seek to evade the filters, spam checking becomes more computationally intensive, resulting in overloaded servers and increased delays.

Our aim is to provide good quality of service for “good” mail in spite of the volumes of junk. The idea is not to stop mail that is identified as junk from reaching users, but to delay it, by delaying its acceptance into the mail server and giving it low priority access to the content scanner or other bottlenecks in the mail system. This

means that false positives of our detection algorithm are not disastrous, because they do not prevent the (eventual) delivery of good email.

We present data collected from two heavily used Internet-facing mail servers from a large corporation, and show that it is relatively simple to predict whether the next message from a particular server will be junk or not. The prediction is based on the types of messages (good/junk) that the server has previously sent, and is fairly accurate (approximately 74–80% of good messages and 93–95% of junk messages are recognized accurately). This prediction can be used to reduce server load by delaying acceptance of messages (using SMTP temporary failures [21]) and to prioritize messages through the mail server. This latter response allows good messages to be processed with minimal delay, even when the server is overloaded.

Using a model of the performance of a mail server, calibrated against our data, we show the effectiveness of this prioritization scheme. For a variety of loading conditions and server performances, the model predicts very low delays (small numbers of seconds) for good mail, compared to the (small numbers of hours) delays observed for a system without prioritization. This not only improves the quality of service, but also means that mail servers will no longer have to grow in processing power at the rate that junk mail increases.

The rest of the paper includes a review of server-based approaches to spam, and an evaluation of these responses given our data. The following sections then describe the prediction method, describe the performance model, and present results for mail transit times. The final sections describe a practical implementation, present initial results, and draw conclusions.

2 Related work

In the SMTP protocol [21] there is a defined point at which the receiving server takes responsibility for delivery of the message. This point is after all the data has been received and acknowledged. Responses to spam at the server can be divided into those before this point (pre-acceptance) and after (post-acceptance). Pre-acceptance responses are preferable, as by stopping the mail they reduce server load. Once the mail has been accepted the server is obliged to do work to deliver it.

Pre-acceptance responses can be divided into three

types: blocking, delaying and tempfailing. Blocking means refusing to accept mail from a particular server, usually identified by IP address (because it is the only part of the SMTP opening dialog that is not easy to forge). Mail is not accepted if it is found on various lists or matches a number of heuristic rules designed to combat spam. For example, a Real-time Blacklist (RBL) is a maintained list of spamming IP addresses, provided by a variety of organizations, and generally accessed over DNS [27]. There are also lists of open relays (poorly configured or compromised machines on the Internet that relay mail indiscriminately), as well as dialup blocks. Other heuristics used include whether the sender's domain is correct, whether the server accepts incoming mails, or whether the sender's address matches the sending domain [29]. Other blocks may be more receiver specific, for example Tantalus [9] blocks a server if it sends too many undeliverable messages.

Blocking is generally accepted and used, and while it is still vulnerable to false positives, there are well known mechanisms to deal with these (e.g. [16]). Unfortunately it takes time to add an address to a blacklist, during which the spammer can send many emails. In addition, our data will show that these blocks are only partially effective, mainly because it is easy for spammers to conform to the heuristics and change their IP address frequently.

The response of delaying is a more direct attack on the resources of the sender of junk mail. Teergrubing [8] is the practice of slowing down the SMTP conversation if a sender tries to send a message to too many recipients (or if the sender's address is on a maintained list). This consumes spammers' resources, but unfortunately also consumes resources on the server. There are many variants on this idea, e.g. more sophisticated teergrubing algorithms (SpamThrottle [38]), and various rate limiting mechanisms built into mail servers, e.g. rate limiting in Postfix [35]. Some delaying tactics work by analyzing the data in the message as it is loading, and slowing the connection if the message looks like spam (e.g. Tar Proxy [32]), and some are like honeypots for spam (e.g. Jackpot Mailserver [5], an open relay that talks slowly to spammers that attempt to use it).

Delaying is a form of rate limiting, and the above techniques limit the number of messages per SMTP connection, or the number of recipients per message. A rate limit on the number of messages per sender is also possible, as offered by IronPort [13], who offer a rating of the trustworthiness of senders, so allowing servers to implement rate limits.

Most of these mechanisms assume that incoming spam is detectable because it is sent at a high rate, and that slowing it down will reduce the amount of spam received. While it is possible for spam to arrive at a high rate, e.g. during a dictionary attack [1, 14] where a spammer bombards a server with messages addressed to random email addresses, our data will show that this generally is not the case. Most spam that is received by our servers is sent by servers from which only small numbers of messages are received, and those messages are not received at particularly high rates. This shows that rate-limiting mechanisms are unlikely to be effective.

The last pre-acceptance response is tempfailing. This is an SMTP control code that means “my server is temporarily unavailable, please try again later” to the sending server [21]. Well-configured mail servers will retry the mail later, but currently most spamming software and email-based viruses do not attempt retries. This is the premise of Greylisting [12], in which a server keeps a list of triples consisting of the sending host’s IP address, sender’s email address and recipient’s email address, and tempfails mail that would generate new triples for some period of time. Since spamming software does not retry, this vastly reduces the amount of spam accepted by a server running Greylisting, while normal mail is subject to some delay. The weakness of this approach is that if (and when) spammers implement retries, this technique will become less effective.

Tempfailing is a useful strategy when used in combination with blacklists. It forces the spammer to maintain the IP address for longer, and reduces the rate that the spammer can send mails. This reduces the number of mails that are sent before the address is added to a blacklist.

Although tempfailing is a useful tool in the fight against spam, our data shows that the Greylisting approach is rather inefficient, delaying a large proportion of good mail, and requiring rather large numbers of triples. We suggest that evidence of spamming can be better obtained by looking at the history of mail sent by a particular server, and that this combined with tempfailing can be effective.

The main post-acceptance response is to look at the content of the mail message, scan it for viruses and examine the text for evidence of spam. Spam filtering can be accomplished by a variety of means, including collections of simple rules [25], Bayesian reasoning [22], and collaborative filtering [6]. It can also be implemented on the client, but is increasingly implemented on the server, often as an add-on to existing scanning for viruses.

None of these filtering mechanisms are infallible, and all suffer from the problem of false positives or incorrectly classified mail. In addition, the filters must be continually updated, as spammers develop new means to evade them [11, 20]. Moreover scanning mechanisms cause extra loading at the server, resulting in delays when the server is heavily loaded.

In this paper we introduce a new post-acceptance response, that of prioritizing the flow of good mail through the mail server, and processing mail suspected of being junk (virus, spam, or undeliverable) at reduced priority. The aim is that in times of heavy system load there will be prompt delivery of good mail, at the expense of delays to other mail.

3 Characteristics of email traffic

To evaluate existing responses and develop new ones, we collected data from the logfiles of two Internet-facing email servers in a large corporation, as detailed in Table 1. Server1 and server2 are the primary and secondary server for a single email domain. These servers ran a virus checker (Sophos [26]), and a spam filter (Spam Assassin [25]), using MailScanner [33], so for each SMTP transaction it is possible to discover from the log files whether received mail contained viruses or was flagged as spam. Server1 and server2 did not store mail for reading or check the addresses of recipients, but forwarded mail to other servers. Undeliverable mail was thus indicated by a failure to deliver to these servers, and was recorded in the logs. The logs also included records from incoming mail blocked because of real-time blacklists and other heuristics. This data gives a good picture of a server’s-eye view of spam.

These servers are listed in public MX lookups, but also receive some mail from other mail servers within the corporation (they are part of an internal mail routing chain). Because the intent of this work was to combat spam entering a corporation, we removed the data corresponding to this “indirect” mail, leaving only the “direct” messages—interactions with other mail servers over the Internet.

Both servers subscribed to real-time blacklists, and performed other checks on the sender’s address before accepting messages. These mechanisms rejected around 34–36% of incoming messages (see Table 2). However this is only partially effective at reducing spam, as

Table 1: Details of the data collected.

server	length (days)	number of messages	number of recipients
server1	69	855,228	1,229,459
server2	69	755,565	1,097,169

Table 2: Effectiveness of real-time blacklists and other blocking responses. While blacklists and other checks block around 34–36% of incoming messages, the accepted mail is still mostly junk, as shown in Table 3.

type	server1		server2	
	number	%	number	%
rbl	256,700	19	207,144	18
open relay	119,161	9	95,536	8
other checks	112,555	8	81,170	7
total blocked	488,416	36	383,850	34
total accepted	855,228	64	755,565	66
total attempts	1,344,960	100	1,139,415	100

Table 3: Breakdown of mail messages by type for each server. The percentages for spam, undeliverable and virus do not sum to 100%, as messages can fall into multiple types. The last two rows show totals of accepted good mail and accepted junk mail, where junk mail is any one of virus, spam or undeliverable.

type	server1		server2	
	number	%	number	%
good	260,348	30	262,941	35
spam	497,554	58	414,234	55
virus	2,749	0.3	2,371	0.3
undeliverable	364,487	43	298,169	39
total accepted good	260,348	30	262,941	35
total accepted junk	594,880	70	492,624	65

around 65–70% of the accepted mail is junk, as shown in Table 3.

Table 3 shows a breakdown of accepted messages. The values for spam and virus reflect messages that were flagged as such in the logs. A message was deemed undeliverable if on the first delivery attempt more than half of the recipients failed. The good messages are those that are left: not virus, spam or undeliverable. The table shows the enormous volume of mail that is junk (65–70% of all accepted mail). This is consistent with other measures of the prevalence of spam [15]. What is also surprising is the number of undeliverable messages. These appear to be mostly spam—of the 364,487 undeliverable messages for server1, only 94,735 (26%) were not classed as spam or virus. A likely cause is “trolling” [12, 1], where spammers send messages to automatically-generated usernames at known domains, hoping that these correspond to genuine email addresses. This figure is far too high to be explained by mistyped addresses, corrupted mailing lists or servers that are offline. Assuming that nearly all of the undeliverable messages are spam suggests that a significant proportion are not being classified correctly by the spam filter. This in turn suggests that the measure for “good” in the table is optimistic: it includes some spam messages. Overall these figures are alarming, as they show how many resources are wasted in passing junk emails through the email system.

Table 4 shows the number of servers that send (only) good mail, (only) junk mail, and a mixture, together with the number of messages sent. A small proportion of servers (11%) send only good mail, and these are responsible for a similar proportion of the messages (11%). By contrast, the overwhelming majority of servers send only junk mail (79%) but generate a relatively small proportion of the messages (48%), implying that each server sends few messages. The mixed class contains data for servers that sent at least one good and one junk mail. This includes those that send mostly good with the occasional junk (e.g. a mistyped address, a false positive spam detection), or those that send mostly junk with the occasional good (e.g. a spammer whose spam occasionally evades spam detection), or a more even mixture that might be obtained from an aggregating mail server (e.g. an ISP). This class is a small proportion of the servers, but generates a large proportion of the messages.

Figure 1 shows the same effect, but broken down by how many messages each server has sent. The figure was constructed by counting how many messages of each type (good/junk) each server sent, and plotting the cumulative percentage of total messages against the num-

Table 4: Number of sending servers and the number of messages that they sent, classified into three groups: those sending only good mail, only junk mail, and a mixture.

type	servers		messages	
	number	%	number	%
server1				
good	24,407	11	97,553	11
junk	178,762	79	413,725	48
mixture	23,416	10	343,950	40
server2				
good	20,508	10	80,082	11
junk	157,467	80	344,669	46
mixture	18,390	9	330,814	44

ber of messages per server. There are three lines, for good, junk and the total.

The line for junk mail shows that most of the junk mail comes from servers that send fewer than 100 messages. In fact 45% of junk mail comes from servers that send fewer than 10 messages. On the other hand, good mail is more likely to come from servers that send many messages. The “total” line is a combination of these two effects, and is more influenced by the junk mail as that makes up the bulk of the messages.

These results suggest that the rate of incoming junk messages from each server is low (it has to be low if they send few messages—10% of junk mail comes from servers that send only one message, for which a rate limit is meaningless). Measurements of local frequency (number of mails per minute) show that some junk mail is sent at high rates, but this is a tiny proportion of the total mail (see Figure 2). There is very little difference between the rates of good and junk senders for the bulk of messages.

This result is perhaps surprising and counter-intuitive, particularly as many of the delaying responses are geared toward reducing rates. This data suggests that, with the possible exception of dictionary attacks (which we did not notice in our data), rate-limiting mechanisms are not particularly effective against junk mail.

The explanation for the shapes of Figures 1 and 2 is a combination of two factors. Firstly, spammers are forced to change their server addresses frequently as addresses are placed on blacklists, so the volume of mail from each spamming server is limited. Secondly, each mail server only sees a sample of the total spam mail sent by

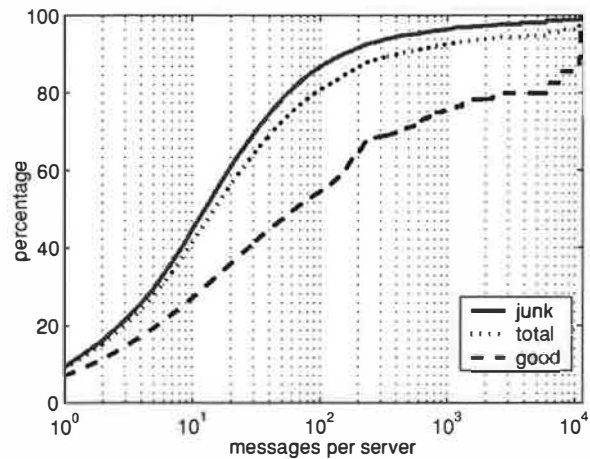


Figure 1: Cumulative histogram of number of messages received from each connecting server, subdivided into number of good, junk, and total. Junk mail tends to be sent by servers sending small numbers of messages (e.g. 50% of junk mail is sent by servers sending fewer than 20 messages). Good mail also has a high proportion from servers sending small numbers of messages, but the effect of large senders is more significant (50% of good mail is sent by servers sending fewer than 60 messages). The data is taken from server1.

any spammer, corresponding to the number of messages on the spammer’s mailing list that are handled by the server. This further reduces the number of messages that are received from each server, and reduces the rate. For servers that handle mail for large domains (e.g. Yahoo) this second factor will be less significant.

Even for good mail, a significant proportion comes from servers that send few messages. This has a bearing on the effectiveness of Greylisting [12], a technique that keeps a record of the triple {sender IP address, sender email address, recipient email address} and only allows mail through without delay if that triple has been observed before. Table 5 shows an evaluation of Greylisting. These results are for 36 days’ worth of data, the standard lifetime of a triple. The maximum number of triples is enormous, but the actual number required to be stored is likely to be a lot lower—Greylisting only keeps a triple for 36 days if a mail with that triple is accepted. Greylisting is very effective against junk mail, with 98% being delayed. On the other hand, the percentage of good mail that is delayed is rather large (40–51%).

However, the fundamental idea behind Greylisting—that

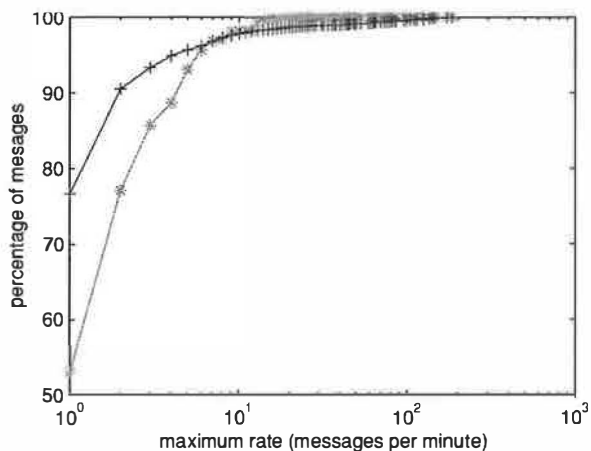
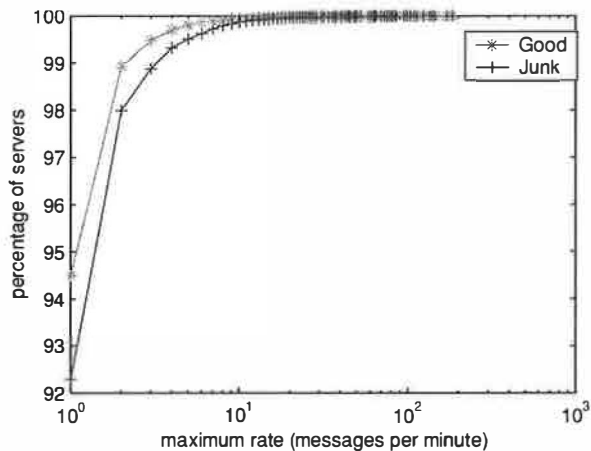


Figure 2: Incoming message rates. The top plot shows the cumulative histogram of incoming message rate, calculated on a per sender basis and normalized. The graph shows that the message rates for good (*) and junk (+) mail are very similar, and most of the mail is sent at a very low rate (92–94% of servers send messages at 1 message/minute). The lower plot shows the cumulative histogram of messages, calculated by assuming that each server sends all its messages at its maximum rate. This is therefore a worst case estimate of the proportion of mail sent at different rates. The bulk of both junk and good messages are sent at rates of less than 10 messages/minute.

Table 5: Results for performance of Greylisting. Greylisting is effective against junk mail, with 98% being delayed, but less effective with good mail, with 40–51% being delayed.

type	number	%
server1		
triples	619,536	
good delayed	84,748	51% of good
junk delayed	444,829	98% of junk
server2		
triples	557,314	
good delayed	69,227	40% of good
junk delayed	376,532	98% of junk

unusual mail is likely to be junk—is a good one, and we explore this further in the following section.

4 Using sending history to predict future mailing behavior

This section develops the idea of using the past history of types of mail sent (good/junk) in order to predict the type of the next message from a server before that mail is accepted. This enables new and interesting responses both pre- and post-acceptance.

The data in the previous section (particularly Table 4) shows that mail servers can be classified based on the mail that they have sent, into those that send junk mails, good mails, and a mixture. This suggests that looking at history might be a good way to predict behavior.

One simple way to represent history is to calculate the proportion of good messages received by the server, where a good message is one that is not classed as spam or virus, and one where more than half of the recipients are delivered on the first delivery attempt. The proportion of good mail P_i can be calculated as

$$P_i = \frac{N_{good}(i)}{N_{total}(i)} \quad (1)$$

where i is the server indexed by IP address, and N_{good} and N_{total} are the number of good and total messages received from server i . When the first message is received there is no data to calculate P_i , so it is initialized to zero. As messages are scanned and delivery attempts

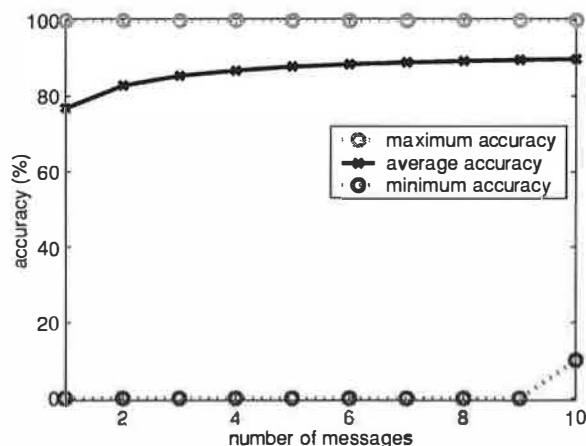


Figure 3: Plot showing the maximum, minimum and average accuracy of the prediction calculated using different numbers of messages, with $r = 0.5$. This data is for all servers that send at least 10 messages. The best case has all predictions correct, the worst 1 correct out of 10. The average goes quickly to near 90%. This result highlights that accuracy improves with history (from 77% with one message to 90% with 10), and also that not much history (< 10 messages) is needed to accurately predict message type.

are made, the values of N_{good} and N_{total} are updated. P_i can be used to predict whether a server is likely to send good or junk mail using a simple threshold r , i.e. if $P_i > r$ the message is predicted to be good and vice versa. As more messages are received, this prediction becomes more accurate (see Figure 3).

Table 6 shows the overall accuracy of this method when $r = 0.5$. It shows that it is important to include all forms of junk mail in order to get an accurate classification; just basing the calculation on spam alone does not give such accurate results. This is most likely due to the fact that combining spam detection and undeliverable mail combines two noisy indicators of spam to give greater accuracy: a proportion of spam evades the spam filter, and many of the addresses on spam mailing lists are incorrect.

Table 6 shows that this measure is remarkably accurate at detecting junk mail (93–95% accuracy), and also good at detecting good mail (74–80%). Some of the errors are unavoidable, as the first mail from every sending server will be predicted to be junk. Although most of the mail sent by new servers is indeed junk, this results in a small number of good messages being misclassified. Out of

Table 6: Accuracy of classification when different data is taken into account. This was calculated by running through the logs and for each sending server evaluating P_i , classifying the message, and updating P_i ready for the next message. Each line in the table includes the effect from the lines above, i.e. the line marked undeliverable is performance for spam + virus + undeliverable. The classification is best when all the types of junk mail are taken into account.

	good		junk	
	number	%	number	%
server1				
spam	242,163	68	462,516	93
virus	239,990	67	464,709	93
undeliverable	192,428	74	565,922	95
server2				
spam	250,496	73	377,212	91
virus	248,516	73	379,112	91
undeliverable	209,309	80	459,432	93

855,228 messages on server1, 226,585 (26%) were the first message from a new server, and of those 33,021 (15%) were good. The other cases where good messages are classed as junk appear from a cursory look to mostly be spam that has evaded the content scanner (for example they have implausible sender email addresses). Thus the errors may be fewer than measured.

The other type of error, where junk messages are classed as good, occurs for an even smaller group. It is probably caused by mistyped addresses, cases where the spam detector over-zealously marks a legitimate mail as spam, or where a sender that has previously sent good mail gets infected by a virus.

Figure 4 shows that the prediction accuracy is insensitive to the value of the threshold r . Varying r in the range 0.1–0.8 gives around 10% variation in the individual accuracies, and virtually no variation in overall accuracy.

The results in Table 6 used the entire 69-day dataset to calculate P_i , however the probability is still remarkably accurate even if only a small part of the history is maintained. Figure 3 shows how even with a very short history of a few messages, the probability measure performs well. This is important as it will allow the quick capture of changes in server behavior, for example if a previously good server starts sending mail infected by a virus.

In addition, good performance can be achieved without

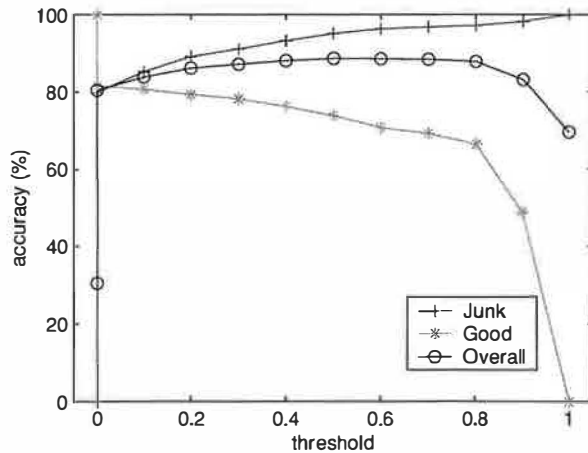


Figure 4: Plot showing the classification accuracy for different values of the threshold r . The overall accuracy is calculated by dividing the total number correct by the total number of messages. A wide range of values of r give good performance. It is interesting that even with $r = 0$, there is a reasonable prediction accuracy. This is because a sender is only classified as good if $N_{good}/N_{total} > r$. For 100% accuracy for good (0% for junk) r needs to be less than zero.

maintaining a record for every sending server. Figure 5 shows the effect on accuracy of maintaining a fixed number of IP records, using a first-in-first-out replacement policy. The total number of unique senders observed is indicated by the vertical line. It is only when the number of addresses stored is around a quarter of the total that the performance starts to fall off.

To summarize, this prediction method is practical to calculate on a real mail server. It requires some state (N_{good}, N_{total}) to be maintained for each server, but that state is either a pair of numbers, or a short (< 10) history of message types. In addition, state can be maintained for a relatively small proportion of the sending servers without sacrificing performance. The amount of state required is far smaller than that required for techniques like Greylisting. The state can also be updated asynchronously as it becomes available. In a real implementation this information will be delayed, however this is unlikely to affect the accuracy of the results significantly (it will only affect updates during overloading).

This method of predicting good and junk mail is particularly powerful because it can be calculated before the message is received. It thus enables both pre- and post-acceptance responses.

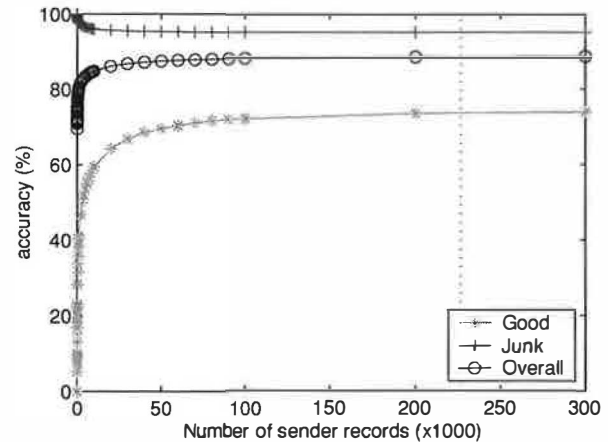


Figure 5: Prediction accuracy for $r = 0.5$ as a function of the number of sender history records maintained, with the vertical line indicating the total number of unique senders in the 69-day dataset.

The rate of false positives is too high to use P_i for blocking mail (in fact if it were used no mail would ever be delivered, because the first mail from every server would be rejected!). It can, however, be used with tempfailing. As mentioned above, the advantage of tempfailing is that it effectively blocks mail if spammers do not retry, and increases the efficiency of blacklists if they do. An example of an appropriate use of tempfailing with the sending history measure is to not accept any mail from a new server for a short period of time (e.g. 4 hours), and from predicted junk-sending servers for a longer period (e.g. 12 hours). This technique would help combat email storms caused by mass mailing viruses: when a previously good machine starts sending the virus, its value of P_i will decrease and eventually its traffic will be tempfailed, so reducing the load on the server. There are, however, other more direct mechanisms to deal with email storms, e.g. [36]. Finally, tempfailing could also be used when the server is heavily loaded, preferentially tempfailing junk mail.

The history measure also enables an exciting post-acceptance response, that of prioritizing good mail through the bottlenecks in the server (often the virus/spam scanner). Instead of all mail being treated equally, good mail can be placed in a high priority queue for the scanner and SMTP processes. The good mail is still scanned, but will travel through the server with minimal delay, even when the server is very heavily loaded by spam or virus attack (both of which are increasingly common).

Table 7: Effect of pre-acceptance responses on good and junk mail. Data from server1.

type	number	percentage
good delayed by 4 hours	33,021	13% of good
good delayed by 12 hours	34,899	13% of good
junk mails rejected	565,922	95% of junk

5 Testing the responses

This section provides some evidence for how much effect the responses described above would have on reducing the amount of spam processed, and on reducing the effect of the volume of junk mail on the flow of good mail through a mail server.

Firstly, we consider the effect of temporarily failing new servers and servers predicted to send junk. Unfortunately it is very difficult to test this just using log data, as the response is to request the remote sending server to retry later, and it is difficult to predict the behavior of the sending server.

A best-case estimate would be to assume that spammers and virus-infected machines do not retry, but that good senders do. Thus out of 855,228 messages accepted by server 1, if a 4 hour delay was used for new senders and a 12 hour delay for predicted junk mail, the effect would be as in Table 7. Only 26% of good mail would be delayed, and as discussed above, a significant proportion of this mail is likely to be misclassified junk. If this sort of system were widely deployed, it is likely that spammers would implement retries. This would cause the amount of junk mail rejected to decline considerably.

It is much easier to predict the effect of post-acceptance responses. We do this by constructing a model of a mail server that allows us to calculate the time taken to process a mail message, under different amounts of loading. We can then alter that model to incorporate prioritization schemes and predict the final performance of the system.

The initial model of the system is shown in Figure 6 (a). This is a generic model of a mail server that includes a mail scanner or filter. The incoming mail is handled by an SMTP process that writes the mail to a local disk or spool q_{MS} , taking time t_{IN} . The mail is then loaded, scanned and placed in a second spool q_{OUT} by the mail scanner, marking the mail accordingly (normally by writing a header), and taking on average t_{SCAN} . The mail is then taken from this second spool and delivered

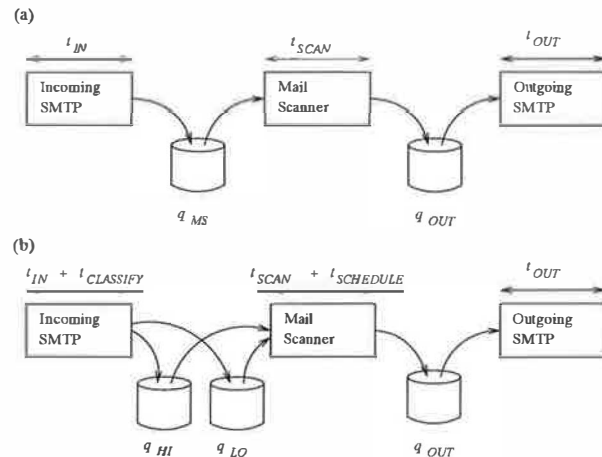


Figure 6: (a) Model of basic mail server. Incoming mail is handled by an SMTP process, before being scanned for viruses and spam by the mail scanner, and being delivered using a second SMTP process. (b) Model of mail server with prioritization. The incoming process places mail in one of two queues depending on the predicted message type. The mail scanner selects messages from the queues using a scheduling algorithm.

by the outgoing SMTP process, taking t_{OUT} . The overall time will be $t_{IN} + t_{SCAN} + t_{OUT}$ plus the time that mails spend on the queues waiting to be processed. As each of the spools is effectively a queue, this system can be analyzed using Queueing Theory [10].

Prioritization can be implemented by having two queues for the mail scanner, one for high priority or good mail (q_{HI}) and the other for low priority or junk mail (q_{LO}). When a mail arrives it is classified into good or junk (this extra processing taking $t_{CLASSIFY}$), and placed into one of these queues. The mail scanner then uses a simple scheduling algorithm to select which message to process next. The simplest algorithm is “absolute priority”, where the scanner always takes from the high priority queue unless it is empty, when it services the low priority queue [19]. The scheduling operation is modelled as taking $t_{SCHEDULE}$. As before, the total time is the sum of the individual times plus the time spent queuing.

We simulated both models using the DEMOS system performance modelling tool [4]. This allows synthetic traffic to be “played” through the model and the system performance evaluated. The parameters of the model (the various service times t_{IN} , t_{SCAN} , t_{OUT}) were estimated from the log data. These parameters are difficult to estimate, because the times in the logs include mes-

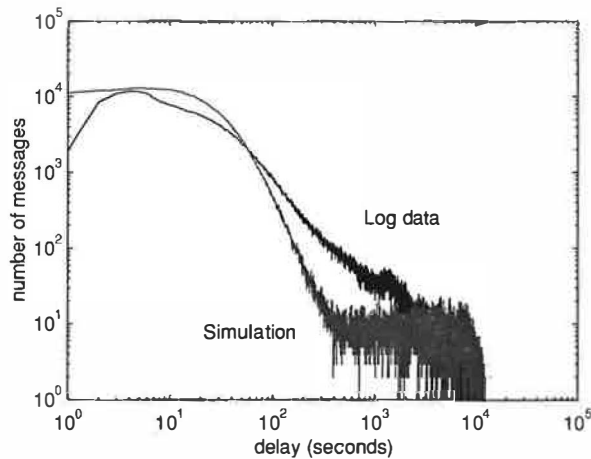


Figure 7: Histogram of message delays. The simulation captures the rough shape of the data, with the bulk of the messages delivered promptly, and a long “tail” of delays. The three service times were modelled as exponential distributions with means of $t_{IN} = 0.02s$, $t_{SCAN} = 7.9s$, $t_{OUT} = 0.08s$. The incoming message rate was modelled as a exponential distribution with mean 7.37s during the day and 11.08s at night.

sages being queued as well as processed. To deal with this we took a lower bound on the actual measurements, assuming that the fastest times corresponded to small or no queues. In addition, we used a facility within DEMOS to specify a probability distribution of parameters rather than a fixed value. The most important parameter is the time to scan a message. This turned out (somewhat surprisingly) not to be particularly sensitive to message size, and was modelled as an exponential distribution. All the other aspects of the model were taken from the log data: the rate and ratio of good/junk were taken from incoming message rates with different rates for day and night; and the traffic was assigned to different priority queues using the probabilities in Table 6, i.e. a good message had an 74% chance of being placed in the high priority queue.

It is important with any model to calibrate it accurately, so that it is possible to believe its predictions. Figure 7 shows the distribution of delays for server1 together with the delays predicted by the model, with parameters as given in the caption. The figure shows that the model fits the data reasonably well.

Having calibrated the model, the effect of the prioritization can be tested. Figure 8 shows the effect of the same parameters applied to the model in Figure 6. The ex-

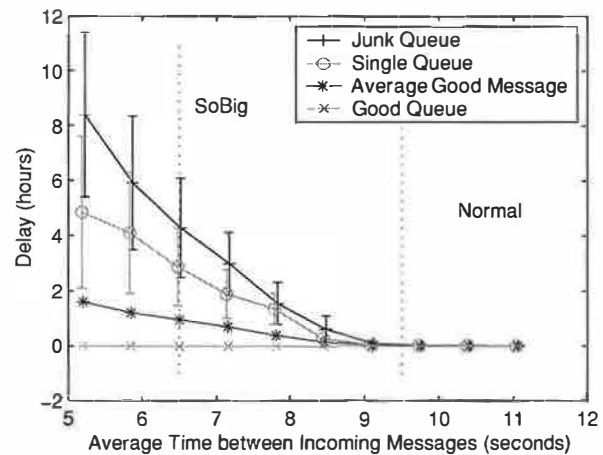


Figure 8: Average delay against traffic loading for the original model (\circ), the good (\times) and the junk ($+$) queue, with error bars showing standard deviations. The two vertical lines indicate traffic loading that is normal, and loading observed during the SoBig.F virus attack. The average delay for good messages ($*$) is also shown, but is not particularly meaningful. Good messages that go in the good queue have small delays, but those wrongly placed in the junk queue will have delays similar to real junk messages. The distribution of delays for good mail is thus bimodal, and not easy to represent with means and standard deviations, especially as the differences in the two distributions are so large, e.g. 6 hours for junk and ≈ 20 seconds for good at 6s between messages.

tra processing times $t_{CLASSIFY}$ and $t_{SCHEDULE}$ were assumed to be small compared to the scan time t_{SCAN} , and were neglected. The plot shows the average delay against the rate of incoming traffic. For low traffic loads the delays are small, because the server is lightly loaded. As the load increases, the delays increase sharply if no prioritization is used. If prioritization is used, the good mail continues to be processed with small delays, but the junk mail is heavily delayed. These traffic loads are common in practice: marked on the diagram are normal loads and loads sampled during the SoBig.F virus attack [31]. For example, during the virus attack, using a conventional arrangement, the mail was delayed by 2.7 hours. With the prioritization scheme, good mail placed in the high priority queue is delayed by only 22 seconds on average, with junk mail delayed by over 4 hours.

Different mail servers have different performance capabilities (different hardware/software configurations), which affect their ability to process mail. Figure 9 shows the behavior of the system with a constant traffic load

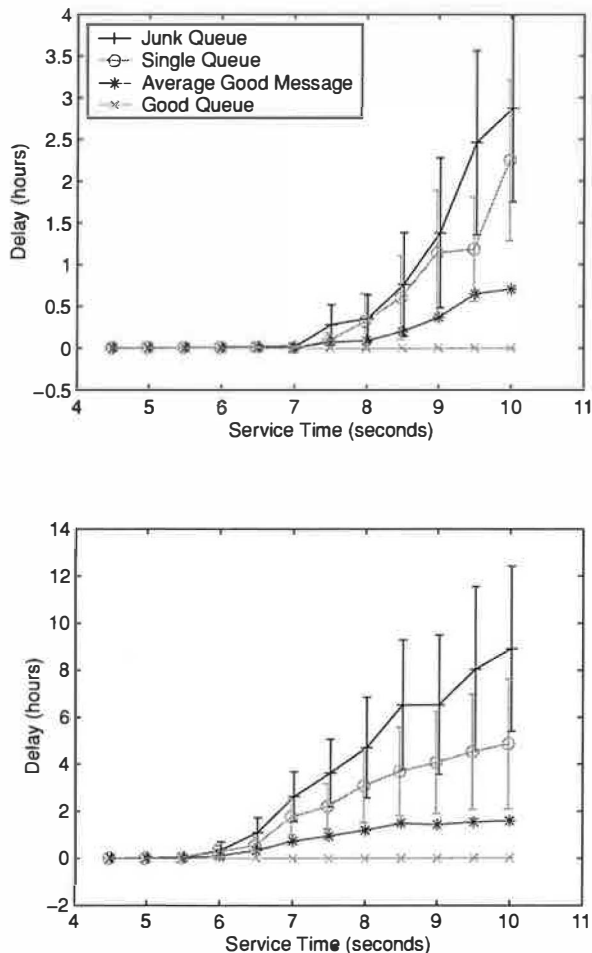


Figure 9: Average delay against server performance (t_{SCAN}) for the original model (\circ), good queue (\times), junk queue ($+$) and average good (*). The top plot corresponds to normal and the lower to SoBig traffic loading. With a single queue, a highly rated server is needed to provide good performance under heavy loading, while with prioritization good mail has consistently low delays even for quite slow servers.

but varying server performance. When the server is over-provisioned the delays are small, but for an under-powered server good mail is still passed with very small delays. This means that mail servers can be sized based on expected normal mail loading, and that their performance will be less sensitive to the amount of junk mail processed. This is a great benefit given the volumes of spam and virus mail that is processed now, let alone the volumes being predicted [18].

Of course, some good mail is wrongly classified as junk, and will have delays similar to the junk mail. However, this delay is only slightly worse than that experienced in the original system for all mail, only a small proportion of mail is affected, and some of that mail should have been classified as spam anyway. In any case it should only be the first legitimate mail from any server that will be heavily delayed.

There are different scheduling schemes that can be used, for example absolute priority, fair share or weighted fair share [7]. We modelled several, but found that the choice of scheduling scheme has little effect on the delivery of good mail: the main effect is on the length of delays to junk mail, with absolute priority giving the longest delays.

To summarize, adding prioritization is extremely effective at ensuring that the bulk of good email is delivered promptly, even when the mail server is very heavily burdened by junk mail.

6 Implementation

Figure 10 shows a sketch of how these schemes could be implemented on an industrial strength mail server. Sendmail [24, 23] and MailScanner [33] are used as examples, but the approach can be generalized to other products.

The basic idea is to mark messages as good or junk in the incoming sendmail process, and dump them in the usual spool directory. The messages are then moved into two queues, which are serviced by two copies of the mail scanner.

When mail is received by the incoming sendmail process, it can be intercepted using the Milter interface [39]. This interface allows access to the mail message and header information, and also allows certain actions like tempfailing or blocking, as well as writing headers and

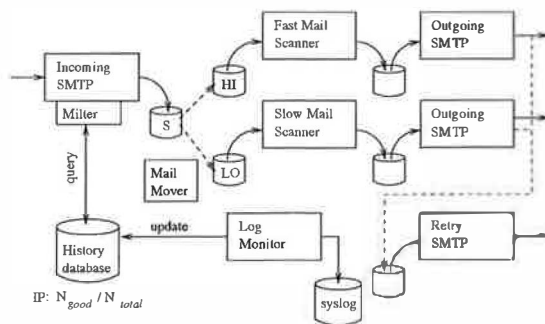


Figure 10: Schematic of implementation on sendmail. Mail is interrupted using the milter interface, and the milter queries the history database to determine for each connection whether to tempfail, or how to mark the message. Mail is placed in the spool *S* as usual, and moved into the appropriate queue by the MailMover process. The copied MailScanner then processes the two queues. Information about the message type is fed back from log files via the LogMonitor process.

changing the message body. The specific milter code for this system would grab the IP address of the sending server, and query the history database to determine the probability that the message was good or junk. The milter would then implement the action required: either to tempfail the request (writing to the database the time when messages from that server would next be accepted), or marking the message as good or junk, for example by writing an extra header into the mail message. If the message is accepted, it is written into the usual mail spool *S*.

The “MailMover” component runs regularly, scanning messages in the spool to determine whether they are good or junk, and moving them into the two queues *HI* and *LO* respectively. Sendmail simply writes the messages into the spool, so it is safe to move the files. (Other servers, e.g. postfix [35] and qmail [3], rely on more complex file information; for these a more sophisticated system would be needed.) Two copies of the MailScanner then process the two queues. These programs would be configured so that the one servicing the high priority queue ran faster than the other.

MailScanner automatically triggers the outgoing SMTP process when it has processed a (configurable) number of messages. Any that are undeliverable are saved in a separate queue, and another sendmail process handles retries. It would be possible to implement prioritization

for this process too, using another MailMover and two retry sendmail processes.

Once the scanner has processed the message, information about the message is available to update the history of the sending server. In addition, once the outgoing sendmail process has attempted to deliver the message, information about undeliverable messages is also available. Since this information is written into the normal syslog, a process that monitors the logs (“LogMonitor” in Figure 10) can recover this information and update the history in the database. It does this by keeping track of message identifiers, and matching the results of the scan/attempted delivery with the IP address of the original sending server.

The database is shown here on a single system; however there is no reason why the information stored should not be shared by many servers. Sharing information is likely to further improve the accuracy of the classification.

7 Preliminary Testing

The system described above was implemented on a linux machine (1.6GHz RedHat 9.1, Sendmail 8.12.11, MailScanner 4.26.28, MySQL 3.23.58). The Milter, MailMover and LogMonitor were all implemented as separate perl processes.

For testing purposes two otherwise identical machines were used, one configured to use the prioritization scheme, and another configured to use a single MailScanner. Figure 11 shows the results of sending the same high load of messages to both machines. The figure clearly shows low delays for good messages even when the load is large.

8 Conclusion

This paper has considered the problem of unacceptable delays in sending email due to servers clogged with unwanted email messages—viruses, spam and undeliverable mail. This is a problem that is significant now, and is likely to become more significant as the volumes of spam and virus-carrying email increase.

We have shown, with reference to empirical data, that existing mechanisms to deal with spam (blacklists, rate

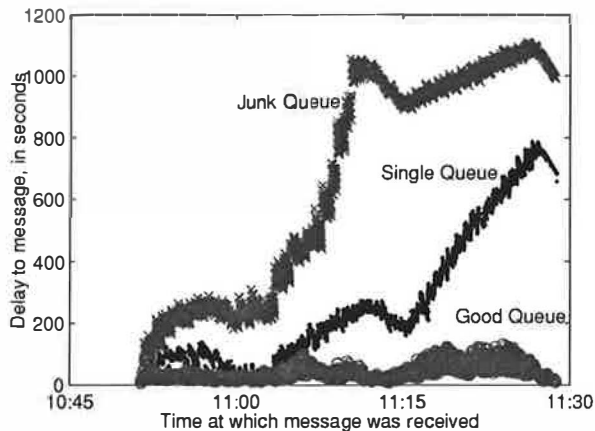


Figure 11: Delays for messages sent through a single queue machine, and the good and junk mail sent through a machine with prioritization. The prioritized good mail is delivered more promptly than it would have been without prioritization.

limiting, greylisting, content filtering) are only partially effective. So, we have developed a system to preserve performance while coping with large volumes of junk mail.

We have shown that a simple prediction of the type (good/junk) of the next message received by a server can be used to delay acceptance of junk mail, and to prioritize good mail through the bottlenecks of the server. The prioritization scheme ensures that most of the good mail is transmitted with small delays, at the expense of longer delays for junk mail. This scheme greatly improves on the performance of current non-prioritized schemes.

We have also argued that this approach is practical, and sketched an implementation on an industry standard mail server that requires no modifications to existing code. Initial tests of this implementation suggest that it works well in practice.

Not all spam classification occurs at the server, and the spam classification at the desktop is often more customizable and accurate. It would be useful to be able to feed this information back to the server. The challenge would be how to achieve this in a practical and secure way. One possible technique would be to write the originating server's IP address in the mail header and allow client software to "update" the mail server with more accurate spam classifications.

In conclusion, while this approach does not stop junk

mail, it should increase the resilience of the mail system, making it better able to cope with overloading from spam and from virus attacks.

Acknowledgements

The authors would like to thank John Hawkes-Reed and Andris Kalnozols for help with data collection and developing the implementation.

References

- [1] 4RSG LLC. What is a dictionary attack? Available at <http://www.filterpoint.com/help/dictionary.html>, 2003.
- [2] Garry Barker. Spam surge slows email. Available at <http://www.theage.com.au/articles/2003/10/13/1065917342993.html>, October 2003. The Age, Technology Section, October 14 2003.
- [3] Dan Bernstein. Qmail. Available from <http://cr.yp.to/qmail.html>, 1997–1998.
- [4] Graham Birtwistle and Chris Tofts. DEMOS. Available from http://www.demos2k.org/demos_ovr.htm, 1979–2002.
- [5] Jack Cleaver. Jackpot Mailserver. Home page: <http://jackpot.uk.net>, January 2003.
- [6] Cloudmark. SpamNet: Learn more: Collaborative SpamFighting. Available at <http://www.cloudmark.com/products/spamnet/learnmore/>, 2003.
- [7] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the ACM SIGCOMM '89 Symposium*, pages 1–12, December 1989.
- [8] Lutz Donnerhacke. Teergrubing FAQ. Available at <http://www.iks-jena.de/mitarb/lutz/usenet/teergrube.en.html>, 1996–2003.
- [9] Brian Gannon. Tantalus v 0.02 – Perl Anti-SPAM Milter. Available at <http://www.linuxmailmanager.com/tantalus.html>, 2003.
- [10] Donald Gross and Carl M. Harris. *Fundamentals of Queueing Theory*. Wiley-Interscience, February 1998.
- [11] Saul Hansell. Internet is losing ground in battle against spam. *The New York Times: Technology section*, 22 April 2003, 2003.
- [12] Evan Harris. The next step in the spam control war: Greylisting. Available at <http://projects.puremagic.com/greylisting/>, July 2003.
- [13] Ironport Systems, Inc. Products and services: Messaging gateway appliances. Available at <http://www.ironport.com/products/>, 2003.

- [14] John Levine, Ray Everett-Church, and Greg Stebben. *Internet Privacy for Dummies*. Wiley Publishing, Inc., August 2002. Paperback edition.
- [15] John Leyden. Spam epidemic gets worse. *The Register*, 55(34331), December 2003. Available at <http://www.theregister.co.uk/content/55/34331.html>.
- [16] Mail Abuse Prevention System LLC. Getting off the MAPS RBL, 2003. <http://mail-abuse.org/rbl/getoff.html>.
- [17] MessageLabs. MessageLabs Monthly View, November 2003. Published on the MessageLabs site, <http://www.messagelabs.com>, November 2003.
- [18] MessageLabs. Spam and viruses hit all time highs in 2003. Published on the MessageLabs site, <http://www.messagelabs.com>, December 2003.
- [19] Glen Nakamura. The GNU C Library – Absolute Priority. Available at <http://www.imodulo.com/gnu/glibc/Absolute-Priority.html>, 2000–2003.
- [20] Martin O’Neal. Corsaire security advisory: Clearswift MAILsweeper MIME attachment evasion issue, March 2003.
- [21] Jonathan B. Postel. RFC 821: Simple Mail Transfer Protocol. Available at <http://www.ietf.org/rfc/rfc821.txt>, August 1982.
- [22] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A Bayesian approach to filtering junk email. Technical Report WS-98-05, AAAI, July 1998. AAAI Workshop on Learning for Text Categorization, Madison, Wisconsin.
- [23] Sendmail Consortium. Welcome to sendmail.org. Freeware version of Sendmail available from <http://www.sendmail.org>, 2001–2003.
- [24] Sendmail Inc. Sendmail site. Commercial version of Sendmail available from <http://www.sendmail.com>, 1999–2003.
- [25] Matt Sergeant. Internet level spam detection and SpamAssassin 2.50. In *Proceedings of the 2003 Spam Conference, Cambridge MA*, January 2003. SpamAssassin is available for download from <http://www.spamassassin.org/index.html>.
- [26] SOPHOS Plc. SOPHOS anti-virus and anti-spam for business. Home page: <http://www.sophos.com/>, 2003.
- [27] Spam-Blockers.com. Email blacklist directory. Available at <http://www.spam-blockers.com/SPAM-blacklists.htm>, 2003.
- [28] Spamhaus Project. Virus and dDoS attacks on Spamhaus. Available at <http://www.spamhaus.org/cyberattacks/index.html>, December 2003.
- [29] Sender policy framework, 2004. <http://spf.pobox.com/>.
- [30] Symantec. Symantec Security Response: W32.Mimail.F@mm. Available at <http://www.symantec.com/avcenter/venc/data/w32.mimail.f@mm.html>, November 2003.
- [31] Symantec. Symantec Security Response: W32.Sobig.F@mm. Available at <http://www.symantec.com/avcenter/venc/data/w32.sobig.f@mm.html>, August 2003.
- [32] TarProxy Project. Tarproxy version 0.29 (dev). Available at <http://sourceforge.net/projects/tarproxy>, June 2003.
- [33] MailScanner team. Mailscanner version 4.24-5. Available at <http://www.mailscanner.info>, October 2003.
- [34] The Turtle Partnership. World data and buzzwords: Messaging. Available at <http://www.turtleweb.com/turtleweb.nsf/listlookup/MarketData>, 2001–2003.
- [35] Wietse Venema. The Postfix home page. Available at <http://www.postfix.org/>, 1998–2003.
- [36] Matthew M Williamson. Design, implementation and test of an email virus throttle. In *Proceedings of ACSAC Security Conference*, Las Vegas, Nevada, December 2003. Available from <http://www.hpl.hp.com/techreports/2003/HPL-2003-118.html>.
- [37] Paul Wood. The convergence of viruses and spam: lessons learned from the SoBig.F experience. Available at <http://www.security.ia.net.au/downloads/sobigwhitepaper.pdf>, 2003.
- [38] Dale Woolridge, James Law, and Moto Kawasaki. The qmail spam throttle mechanism. Available at <http://spamthrottle.qmail.ca/man/qmail-spamthrottle.5.html>, 2002–2003.
- [39] Charles Ying. Sendmail::Milter. Available from <http://sendmail-milter.sourceforge.net/>, 2003.

Redundancy Elimination Within Large Collections of Files

Purushottam Kulkarni
University of Massachusetts
Amherst, MA 01003
purukulk@cs.umass.edu

Fred Douglass Jason LaVoie John M. Tracey
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{fdouglass,lavoie,tracey}@us.ibm.com

Abstract

Ongoing advancements in technology lead to ever-increasing storage capacities. In spite of this, optimizing storage usage can still provide rich dividends. Several techniques based on delta-encoding and duplicate block suppression have been shown to reduce storage overheads, with varying requirements for resources such as computation and memory. We propose a new scheme for storage reduction that reduces data sizes with an effectiveness comparable to the more expensive techniques, but at a cost comparable to the faster but less effective ones. The scheme, called *Redundancy Elimination at the Block Level* (REBL), leverages the benefits of compression, duplicate block suppression, and delta-encoding to eliminate a broad spectrum of redundant data in a scalable and efficient manner. REBL generally encodes more compactly than compression (up to a factor of 14) and a combination of compression and duplicate suppression (up to a factor of 6.7). REBL also encodes similarly to a technique based on delta-encoding, reducing overall space significantly in one case. Furthermore, REBL uses *super-fingerprints*, a technique that reduces the data needed to identify similar blocks while dramatically reducing the computational requirements of matching the blocks: it turns $O(n^2)$ comparisons into hash table lookups. As a result, using super-fingerprints to avoid enumerating matching data objects decreases computation in the resemblance detection phase of REBL by up to a couple orders of magnitude.

1 Introduction

Despite ever-increasing capacities, significant benefits can still be realized by reducing the number of bytes needed to represent an object when it is stored or sent. The benefits can be especially great for mobile devices with limited storage or bandwidth; reference data (data that are saved permanently and accessed infrequently); e-mail, in which large byte sequences are commonly repeated; and data transferred over low-bandwidth or congested links.

Reducing bytes generally equates to eliminating unneeded data, and there are numerous techniques for reducing redundancy when objects are stored or sent. The most longstanding example is *data compression* [12],

which eliminates redundancy internal to an object and generally reduces textual data by factors of two to six. *Duplicate suppression* eliminates redundancy caused by identical objects which can be detected efficiently by comparing hashes of the objects' content [3]. *Delta-encoding* eliminates redundancy of one object relative to another, often an earlier version of the object by the same name [15]. Delta-encoding can in some cases eliminate an object almost entirely, but the availability of base versions against which to compute a delta can be problematic.

Recently, much work has been performed on applying these techniques to pieces of individual objects. This includes suppressing duplicate pieces of files [7, 8, 17, 20] and web pages [22]. Delta-encoding has also been extended to pairs of files that do not share an explicit versioning relationship [6, 9, 18]. There are also approaches that combine multiple techniques; for instance, the *vcdiff* program not only encodes differences between a "version" file and a "reference" file, it compresses redundancy within the version file [11]. Delta-encoding that simultaneously compresses is sometimes called "delta compression" [1].

In fact, no single technique can be expected to work best across a wide variety of data sets. There are numerous trade-offs between the *effectiveness* of data reduction and the resources required to achieve it, i.e. its *efficiency*. The relative importance of these metrics, effectiveness versus efficiency, depends on the environment in which techniques are applied. Execution time for example, which is an important aspect of efficiency, tends to be more important in interactive contexts than in asynchronous ones. In this paper, we describe a new data reduction technique that achieves comparable effectiveness to current delta-encoding techniques but with greater efficiency. It simultaneously offers better effectiveness than current duplicate suppression techniques at moderately higher cost.

We argue that performing comparisons at the granularity of files can miss opportunities for redundancy elimination, as can techniques that rely on large contiguous pieces of files to be identical. (Henson's study of issues relating to comparing blocks by hashes of their content made a similar argument [10].) Instead, we consider

what happens if some of the above techniques are further combined. Specifically, we describe a system that supports the union of three techniques: compression, elimination of identical content-defined chunks, and delta-compression of similar chunks. We refer to this technique as Redundancy Elimination at the Block Level (REBL). The key insight of this work is the ability to achieve more effective data reduction by exploiting relationships among similar blocks, rather than only among identical blocks, while keeping computational and memory overheads comparable to techniques that perform redundancy detection with coarser granularity.

We compare our new approach with a number of baseline techniques, which are summarized here and described in detail in the next section:

Whole-file compression. With whole-file compression (WFC), each file is compressed individually. This approach gains no benefit from redundancy across files, but it scales well with large numbers of files and is applicable to storage and transfer of individual files.

Compressed tar. Joining a collection of files into a single object which is then compressed has the potential to detect redundancy both within and across files. This approach tends to find redundancy across files only when the files are relatively close to one another in the object. We abbreviate this technique TGZ, for `tar+gzip`, the combination we used.

Block-level duplicate detection. There are a number of approaches to identifying identical pieces of data across files more generally. These include using fixed-size blocks [20], fixed-size blocks with rolling checksums [8, 23, 29], and content-defined (and therefore variable-sized) chunks [7, 17].

Delta-encoding using resemblance detection.

Resemblance detection techniques [4] can be used to find similar files, with delta compression used to encode them effectively [9, 18].

There are also cases where effectiveness is dramatically improved by combining multiple techniques, such as adding compression to block-level or chunk-level duplication detection.

The remainder of this paper is organized as follows: Section 2 describes current techniques and their limitations. Details of the REBL technique are presented in Section 3. Section 4 describes the data sets and methodology used to evaluate REBL. Section 5 presents an empirical evaluation of REBL and several other techniques. Section 6 concludes.

2 Background and Related Work

We discuss current techniques in Section 2.1 and elaborate on their limitations in Section 2.2.

2.1 Current Techniques

A common approach to storing a collection of files compactly is to combine the files into a single object, which is then compressed on-the-fly. In Windows™, this function is served by the family of *zip* programs, though they do not necessarily identify inter-file redundancy in addition to intra-file redundancy. In UNIX™, files can be combined using *tar* with the output compressed using *gzip* or another compression program. However, TGZ does not scale well to extremely large file sets. Access to a single file in the set can potentially require the entire collection to be uncompressed. Furthermore, traditional compression algorithms maintain a limited amount of state information. This can cause them to miss redundancy between sections of an object that are distant from one another thus reducing their effectiveness. Historically the window used to detect redundancy is small, for instance 32 KB, but at least one new compression program uses memory-mapped I/O and increased state to find repeated substrings across hundreds of MB [24].

There are two general methods for compressing a collection of files with greater effectiveness and scalability than TGZ. One involves finding identical chunks of data within and across files. The other involves effective encoding of differences between files.

2.1.1 Duplicate Elimination

Finding identical files in a self-contained collection is straightforward through the use of strong hashes of their content. For example, Bolosky et al. have described a system to save only one instance of duplicate files in a Windows file system [3]. Mogul et al. have described a method for computing checksums over web resources (HTML pages, images, etc.) and eliminating retrieval of identical resources, even when accessed via different URIs [16].

Suppressing redundancy within a file is also important. One simple approach is to divide the file into fixed-length blocks and compute a checksum for each. Identical blocks are detected by searching for repeated checksums. The checksum algorithm must be “strong” enough to decrease the probability of a collision to an negligible value. SHA-1 [26] (“SHA”) is commonly used for this purpose.

Venti [20] is a network-based storage system intended primarily for archival purposes. Each stored file is broken into fixed-sized blocks, which are represented by their SHA hashes. As files are incrementally stored, duplicate blocks, indicated by identical SHA values, are stored only once. Each file is represented as a list of SHA hashes which index blocks in the storage system.

Another algorithm used to minimize the bandwidth required to propagate updates to a file is *rsync* [23, 29]. With *rsync*, the receiver divides its out-of-date copy of

a file into fixed-length blocks, calculates two checksums for each block (a weak 32-bit checksum and a strong 128-bit MD4 checksum), and transmits the checksums to the sender to inform it which blocks the receiver possesses. The sender calculates a 32-bit checksum along a fixed-length window that it slides throughout the file to be sent. If the 32-bit checksum matches a value sent by the receiver, the sender confirms that the receiver already possesses the corresponding block by computing and comparing the 128-bit checksum. Use of a rolling checksum over fixed-sized blocks has recently been extended to large collections of files regardless of name [8]. We refer to this as the *SLIDINGBLOCK* approach, which is one of the techniques against which we compare REBL below.

One can maintain a large replicated collection of files in a distributed environment using a technique similar to *SLIDINGBLOCK* [27]. Suel et al. point out two main parameters for *rsync* performance: block size and location of changes within the file. To enhance the performance of *rsync*, they propose a multi-phase protocol in which the server sends hashes to the client and client returns a bitmap indicating the blocks it already has (similar to *rsync*). In this approach, the server uses the bitmap of existing blocks to create a set of reference blocks used to encode all blocks not present at the client. The delta sent to the client by the server is used in conjunction with blocks in the bitmap to recreate the original file. This technique has some similarity to Spring and Wetherall's approach to finding redundant data on a network link by caching "interesting" fingerprints of sliding windows of data and then finding the fingerprints in a cache of past transmissions [25].

The Low-Bandwidth File System (LBFS) [17] is a network file system designed to minimize bandwidth consumption. LBFS divides files into *content-defined chunks* using Rabin fingerprints [21]. Fingerprints are computed over a sliding window; a subset of possible fingerprint values denotes chunk boundaries, with the subset determining a probabilistic average chunk size. LBFS also imposes a minimum and maximum chunk size, irrespective of fingerprint values.

LBFS computes and stores an SHA hash for each content-defined chunk stored in a given file system. Before a file is transmitted, the SHA values of each chunk in the file are sent first. The receiver looks up each hash value in a database of values for all chunks it possesses. Only chunks not already available at the receiver are sent; chunks that are sent are compressed. Content-defined chunks have also been used in the web [22] and backup systems [7]. We refer to the overall technique of eliminating duplicate content-defined chunks and compressing remaining chunks as CDC, and we compare REBL with this combined technique in the

evaluation section below.

2.1.2 Delta-encoding and File Resemblance

A second general approach to compressing multiple data objects is delta-encoding. This approach has been used in many applications including source control [28], backup [1], and web retrieval [14, 15]. Delta-encoding has also been used on web pages identified by the similarity of their URIs [6].

Effective delta-encoding relies on the ability to detect similar files. Name-based file pairing works only in very limited cases. With large file sets, the best way to detect similar files is to examine the file contents. Manber [13] discusses a basic approach to finding similar files in a large collection of files. His technique summarizes each file by computing a set of polynomial-based fingerprints; the similarity between two files is proportional to the fraction of fingerprints common between them. Rabin fingerprints have been used for this purpose in numerous studies. Broder developed a similar approach [4], which he extended with an heuristic to coalesce multiple fingerprints into *super-fingerprints*. A single matching super-fingerprint implies high similarity, allowing similarity detection to scale to very large file sets such as web search engines [5].

While these techniques allow similar files to be identified, only recently have they been combined with delta-encoding to save space. Douglass and Iyengar describe "Delta-Encoding via Resemblance Detection" (DERD), a system that uses Rabin fingerprints and delta-encoding to compress similar files [9]. The similarity of files is based on a subset of all fingerprints generated for each file. Ouyang et al. also study the use of delta compression to store a collection of files [18]; their approach to scalability is discussed in the next subsection.

2.2 Limitations of Current Techniques

Duplicate elimination exploits only files, blocks or chunks that are exactly the same. Thus, a version of a file that has many minor changes scattered throughout sees no benefit from the CDC or *SLIDINGBLOCK* techniques. Section 5.1 includes graphs of the overlap of fingerprints in CDC chunks that indicate how common this issue can be.

DERD uses delta-encoding, which eliminates redundancy at fine granularity when similar files can be identified. Resemblance detection using Rabin fingerprints is more efficient than the brute force approach of delta-encoding every possible pair of files. A straightforward approach to identifying similar files is to count the number of files that share even a single fingerprint with a given file. However, repeating this for every fingerprint of every file results in an algorithm with $O(n^2)$ complex-

ity in the worst case, where n is the number of files.¹ For large file sets, run time is dominated by the number of pairwise comparisons and can grow quite large even if the time for each comparison is small. DERD's performance therefore does not scale well with large data sets.

The computational complexity of delta-encoding file sets motivates cluster-based delta compression [18]. With this approach, large file sets are first divided into clusters. The intent is to group files expected to bear some resemblance. This can be achieved by grouping files according to a variety of criteria including names, sizes, or fingerprints. (Douglass and Iyengar used name-based clusters to make the processing of a large file set tractable in terms of memory consumption [9]; similar benefits apply to processing time.) Once files are clustered, the techniques described above can be used to identify good candidate pairs for delta-encoding within each cluster. Clustering reduces the size of any file set to which the $O(n^2)$ algorithm described above is applied. When applied over all clusters, the technique results in an approximation to the optimal delta-encoding. How close this approximation is depends on the amount of overlap across clusters and is therefore extremely data-dependent.

Another important issue is that DERD does not detect matches between an encoded object and pieces of multiple other objects. Consider for example, an object A that consists of the concatenation of objects B - Z . Each object B - Z could be encoded as a byte range within A , but DERD would likely not detect any of the objects B - Z as being similar to A . This is due, in part, to the decision to represent each file by a fixed number of fingerprints regardless of file size. Because Rabin fingerprint values are uniformly distributed, the probability of a small file's fingerprints intersecting a large containing file's fingerprints is proportional to the ratio of their sizes. In the case of 25 files contained within a single 26th file, if the 25 files are of equal size but contain different data, each will contribute about $\frac{1}{25}$ of the fingerprints in the container. This makes detection of overlap unlikely.

The problem arises because of the distinction between resemblance and containment. Broder's definition of *containment* of B in A is the ratio of the intersection of the fingerprints of the two files to the fingerprints in B , i.e. $\frac{F(A) \cap F(B)}{F(B)}$ [4]. When the number of fingerprints for a file is fixed regardless of size, the estimator of this intersection no longer approximates the full set. On the other hand, deciding there is a strong resemblance between the two is reasonably accurate, because for two documents to resemble each other, they need to be of similar size.

Finally, it is possible that extremely large data sets would not lend themselves to "compare-by-hash" be-

cause of the prospect of an undetected collision [10]: hashes can be collision-resistant but there will be collisions given enough inputs. In a system that is deciding whether two local objects are identical, a hash can be used to find the two objects before expending the additional effort to compare the two objects explicitly. Our data sets are not of sufficient scale for that to pose a likely problem, so we did not include this extra step. While we chose to follow the protocols of past systems, explicit comparisons could easily be added.

3 REBL Overview

We have designed and implemented a new technique that applies aspects of several others in a novel way to attain benefits in both effectiveness and efficiency. This technique, called Redundancy Elimination at the Block Level (REBL), includes features of compression, CDC, and DERD. It divides objects into content-defined chunks, which are identified using SHA hashes. First duplicate chunks are removed, and then resemblance detection is performed on each remaining chunk to identify chunks with sufficient redundancy to benefit from delta-encoding. Chunks not handled by either duplicate elimination or resemblance detection are simply compressed. A more detailed description of REBL appears in Section 4.1.

Key to REBL's ability to achieve efficiency comparable to CDC, instead of suffering the scalability problems of DERD, are optimizations that allow resemblance detection to be used more effectively on chunks rather than whole files. Resemblance detection has been optimized for use in Internet search engines to detect nearly identical files. The optimization consists of summarizing a set of fingerprints into a smaller set of *super-fingerprints*, possibly a single super-fingerprint. Objects that share even a single super-fingerprint are extremely likely to be nearly identical [5].

Optimized resemblance detection works well for Internet search engines where the goal is to detect documents that are nearly identical. Detecting objects that are merely similar enough to benefit from delta-encoding is harder. We hypothesized that applying super-fingerprints to full files in DERD would significantly improve the time needed to identify similar files but would also dramatically reduce the number of files deemed similar, resulting in lower savings than the brute force technique that counts individual matching fingerprints [9]. In practice, we found using the super-fingerprint technique with whole files works better than we anticipated, but it is still not the most effective approach (see Section 5.1.4 for details).

In contrast, REBL can benefit from the optimized resemblance detection because it divides files into chunks and looks for near duplicates of each chunk separately.

Data set	Size (MB)	# files	# chunks	
			1 KB	4 KB
Slashdot	38.37	885	21,991	11,629
Yahoo	27.77	3,850	28,542	8,632
Emacs	106.60	5,490	70,640	24,960
MH	602.10	93,867	421,501	203,518
Users	6625.43	185,722	3,949,780	1,367,619

Table 1: Details of data sets used in our experiments. 1 KB and 4 KB are the targeted average chunk sizes. For 1-KB averages, the minimum chunk size is set to 512 bytes; for 4-KB averages, it is set to 1 KB. The maximum is 64 KB.

This technique can potentially sacrifice some “marginal” deltas that would save some space. We quantify this sacrifice by comparing the super-fingerprint approach with the DERD technique that enumerates the best matches from exact fingerprints.

3.1 Parameterizing REBL

REBL’s performance depends on several important parameters. We describe the parameters and their default values here and provide a sensitivity analysis in Section 5.

Average chunk size. Smaller chunks detect more duplicates but compress less effectively; they require additional overhead to track one hash value and numerous fingerprints per chunk; and they increase the number of comparisons and delta-encodings performed. We found 1 KB to be a reasonable default, though 4 KB improves efficiency for large data sets with large files. Throughout this paper, references to REBL with a specific chunk size refer to a probabilistic average chunk size.

Number of fingerprints per chunk. The more fingerprints associated with a chunk, the more accurate the resemblance detection but the higher the storage and computational costs. Douglass and Iyengar used 30 fingerprints, finding no substantial difference from increasing that to 100 [9].

Number of fingerprints per super-fingerprint. With super-fingerprints, a given number of base fingerprints are distilled into a smaller number of super-fingerprints. We use 84 fingerprints, which are grouped into 3-42 super-fingerprints.

Similarity threshold. How many fingerprints (or super-fingerprints) must match to declare two chunks similar? If the threshold is fixed, how important is it to find the “best” match rather than any adequate match? Ouyang et al. addressed this by finding adequate matches rather than best matches [18]; Douglass and Iyengar did a more computationally expensive but more

precise determination [9]. We take a middle ground by approximating the “best” match more efficiently via super-fingerprints. A key result of our work is that using a sufficiently large number of fingerprints per super-fingerprint allows any matching chunk to be used rather than having to search for a good match. This results in nearly the same effectiveness but with substantially better efficiency (see Section 5.1.2).

Base minimization. Using the best base against which to delta-encode a chunk can result in half the chunks serving as reference blocks.² Allowing approximate matches can substantially increase the number of version blocks encoded against a single reference block, thereby improving overall effectiveness (see Section 5.3.2).

Shingle size. Rabin fingerprints are computed over a sliding window called a shingle and used for two purposes. First, CDC uses specific values of Rabin fingerprints to denote a chunk boundary. Second, DERD uses them to associate features with each chunk. A shingle should be large enough to generate many possible substrings, which minimizes spurious matches, but it should be small enough to keep small changes from affecting many shingles. Common values in past studies like DERD have ranged from four to twenty bytes [9, 18]. We used a default of twelve bytes but found no consistent trend other than a negative effect from sizes of four or eight bytes in one of the data sets (see Section 5.3.3).

4 Data Sets and Methodology

We used several data sets to test REBL’s effectiveness and efficiency. Table 1 lists the different data sets, giving their size, the number of files, and the number of content-defined chunks with the targeted average chunk-size set to 1 KB and 4 KB.

The Slashdot and Yahoo data sets are web pages that were downloaded and saved, as a system such as the Internet Archive [2] might archive web pages. Slashdot represents multiple pages downloaded over a period of about a day, wherein different pages tend to have numerous small changes corresponding mostly to updated counts of user comments. (While the Internet Archive would not currently save pages with such granularity, an archival system might if it could do so efficiently.) As the smallest data set, Slashdot is used below in several cases where there are large numbers of experiments with varying parameters to keep the total execution time within reason. Yahoo represents a number of different pages downloaded recursively at a single point in time. Emacs contains the source trees for GNU Emacs 20.6 and 20.7. The MH data set refers to individ-

ual files consisting of entire e-mail messages. Finally, *Users* is the contents of one partition of a shared storage system within IBM, containing data from 33 users totaling nearly 7 GB.

REBL is intended for much larger data sets than the ones presented here. However, the analysis was implemented using in-memory data structures based on the GNU C++ Standard Template Library, and as a result the application's virtual address space places limits on the metadata (particularly matching pairs of fingerprints) kept in memory. The results presented below demonstrate the scalability issues mentioned previously, and lead us to believe that one can extrapolate to larger data sets once out-of-memory data structures are used. Furthermore, one can vary parameters such as block size and the number of super-fingerprints per block to keep the meta-data requirements low. In particular, the *Users* data set is an order of magnitude larger than the next-largest data set, containing many large files, so the REBL analysis of it is done with an average chunk size of 4 KB rather than 1 KB.

4.1 REBL Evaluation

To evaluate REBL, we first read each file in the data set sequentially, break it into content-defined chunks and generate the Rabin fingerprints and SHA hash values for each chunk. The hashes and fingerprints are stored in Berkeley DB format; each chunk has a file name, length, offset, and fingerprints associated with it. In this stage, chunks with the same SHA hash value as earlier chunks require no additional processing, because they are suppressed by the CDC duplicate detection mechanism.

A separate application reads the fingerprints and chunk information to perform REBL or DERD analysis. First, it computes super-fingerprints from the fingerprints, given a specific ratio of fingerprints per super-fingerprint (with a ratio of one, this step would be skipped). At this point, we have the option of *FirstFit* or *BestFit*.

- To do *FirstFit*, we pick a candidate reference chunk and encode all other chunks that share a super-fingerprint with it; an associative array makes pairwise matching efficient. We then iterate over the remaining candidate reference chunks, performing the same operation, ignoring any chunks that have already been encoded or used as a reference. The chunks are analyzed in order of insertion into the system; i.e., the first file read by the system will be broken into one or more chunks, each of which is likely to serve as a reference version for other chunks with many fingerprints in common, and chunks from later files will be increasingly likely to have already been encoded.
- To do *BestFit*, we sort the chunks according to the

greatest number of matching fingerprints with any other chunk. Each candidate reference chunk is then processed to determine which other potential version chunks have at least a threshold number of super-fingerprints in common with it. The threshold is a specified fraction of the best match found for that chunk (see Section 5.3.2). Again, each chunk is used as either a reference against which one or more version chunks are encoded, or as a version. *BestFit* suffers from quadratic complexity in processing time, as a function of the number of chunks, as well as substantially greater memory usage³ than *FirstFit*.

Next we compress any remaining chunks that have not already been delta-encoded, including the reference chunks.

Finally, each of the files in the data set is compressed to determine if compressing the entire file is more effective than eliminating duplicate blocks and delta-encoding similar blocks. If so, the WFC size is used instead. We found that CDC in the absence of WFC was much less effective than the combination of the two, but adding WFC to REBL usually made only a small difference because most chunks already benefitted from delta-encoding (see Section 5.2).

One technique we did not explicitly evaluate is duplication detection using fixed-size blocks, like that performed by Venti [20]: we worked under the assumption that CDC would be preferable to fixed-size blocks. Other studies that compare the two techniques head-to-head have found that CDC frequently compresses better, at the cost of increased computation [8, 19].

5 Empirical Results

This section provides an empirical evaluation of several data reduction techniques with an emphasis on REBL. For REBL, we study the effect of parameters such as the average chunk size, the number of super-fingerprints, the similarity threshold above which delta-encoding is applied, and the shingle size. The techniques are compared along primarily two-dimensions: *effectiveness* (space savings) and *efficiency* (run time).

The techniques evaluated in at least one scenario include:

- TGZ
- whole-file compression (WFC)
- per-block compression (PBC)
- CDC (includes PBC)
- CDC with WFC
- SLIDINGBLOCK
- REBL with WFC
- DERD with WFC

In cases where WFC is used in conjunction with another

technique, this means that WFC is used instead of the other technique if it is found to be more effective on a particular file.

Our experiments were performed on an unmodified RedHat Linux 2.4.18-10 kernel running on an IBM eServer xSeries 360 with dual 1.60 GHz Pentium Xeon processors, 6 GB RAM (2×2 GB plus 2×1 GB), and three 36 GB 10k-RPM SCSI disks connected to an IBM Netfinity ServeRAID™ 5 controller. All data sets resided in an untuned ext3 file system on local disks. Although an SMP kernel was used, our tests were not optimized to utilize both processors. All times reported are the sums of user and system time as reported by *getrusage*.

5.1 REBL Hypotheses

This section presents empirical results to support the rationale behind the combination of chunk-level delta-encoding and super-fingerprints.

5.1.1 Chunk Similarity

As discussed in Section 2.1, CDC systems such as LBFS [17] compute SHA hashes of content-defined chunks and use the hash value to detect duplicates. A potential limitation of this approach is that chunks with slight differences get no benefit from the overlap. For REBL to be more effective than CDC, there must be a substantial number of chunks that are similar but not identical.

Figure 1 plots a cumulative distribution of the fraction of chunks that match another chunk in a given number of fingerprints. The graph shows results for the *Slashdot* and *Yahoo* data sets with 84 fingerprints per chunk and shows curves corresponding to average chunk sizes of 1 KB, 4 KB, and whole files. Whole files correspond to an infinitely large average chunk size, which is similar to DERD. All chunks match another chunk in at least 0 fingerprints, so each curve meets the 0 value on the x -axis at $y = 1$. The rightmost points on the graph (depicted as $x=85$) show the fraction of chunks that are duplicated; smaller chunks lead to greater effectiveness for CDC because they allow duplicate content to be detected with finer granularity. Between these extremes, more of the smallest chunks match other chunks in the greatest number of features. However, any chunks that are not exact duplicates but match many fingerprints are “missed” by CDC, but they are potentially usable by REBL for delta-encoding and result in improved space savings. A good heuristic for expecting improvement from delta-encoding is to match at least half the fingerprints [9].

5.1.2 Benefits of Super-fingerprints

Next we look at the use of a smaller number of super-fingerprints to approximate a larger number of fingerprints. As discussed in Section 3, super-fingerprints are generated by combining fingerprints. For instance, 84 fingerprints can be clustered into groups of six to form 14 super-fingerprints. To generate super-fingerprints, REBL concatenates fingerprints and calculates the corresponding MD5 value of the resulting string.⁴

Note that the clustering of fingerprints into super-fingerprints is necessary to make the *FirstFit* variant viable. One could delta-encode any pair of chunks that matched a single fingerprint, but unless the shingle size is quite high, there is little assurance of commonality between the chunks. On the other hand, if two chunks share a specific set of fingerprints, the larger the set, the greater the likelihood of a significant overlap [5].

Figure 2 plots the cumulative distribution of the number of chunks that match another in at least a threshold fraction of fingerprints or super-fingerprints. The data sets used are *Slashdot* and *MH*, with 84 fingerprints, 1-KB average chunks, and 21, 14, 6 and 4 super-fingerprints per chunk. (The curve for 84 fingerprints on the *Slashdot* data set corresponds to the 1-KB curve for *Slashdot* in the preceding figure.) The results indicate that lowering the threshold for similarity between chunks results in more chunks being considered “similar.” The results for super-fingerprints follow a similar trend as for regular fingerprints. A useful observation from both data sets is that we can select a threshold value for super-fingerprints that corresponds to a higher number of matching fingerprints. For example, in Figure 2(a), a threshold of one out of four (25%) super-fingerprints is approximately equivalent to a threshold of 73 out of 84 (87%) fingerprints. Using super-fingerprints therefore decreases REBL’s execution time by reducing the number of comparisons, as the next subsection describes.

5.1.3 FirstFit and BestFit Variants

As discussed in Section 4.1, REBL has two variants, *BestFit* and *FirstFit*. In this section, we compare them by contrasting relative effectiveness and efficiency.

Figure 3(a) plots the sizes of the *MH* and *Slashdot* data sets, encoded using the *FirstFit* and *BestFit* variants, and reported relative to the original data set sizes. The experiment uses 84 fingerprints per chunk and two chunking methods, whole files (DERD) and an average chunk size of 1 KB. In this subsection we consider only the 1-KB chunks, discussing later how this compares to other sizes. The figure demonstrates the effect of varying the number of super-fingerprints from three to 84 (with 84 meaning there is no clustering). Both *FirstFit* and *BestFit* have comparable encoding sizes for up

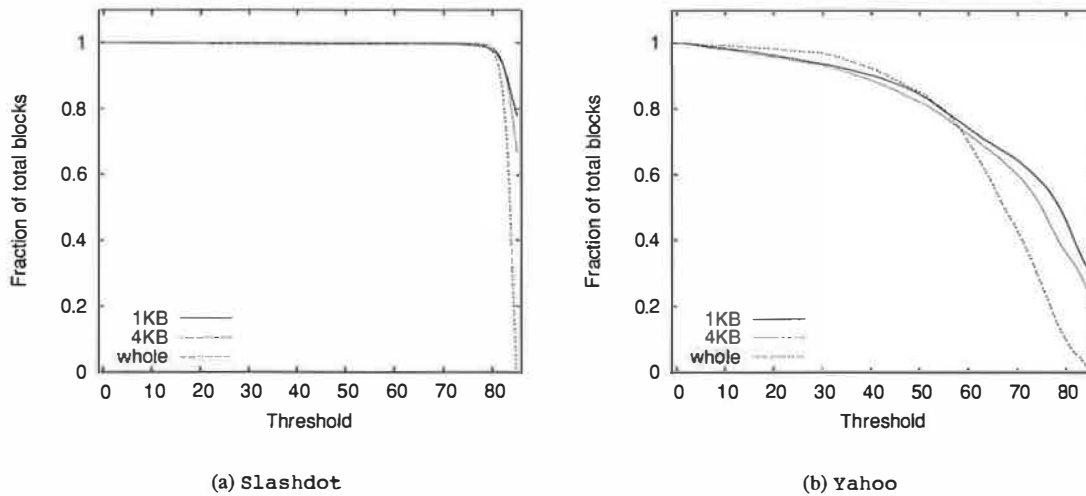


Figure 1: Cumulative distribution of the fraction of similar chunks or files with at least a given number of maximally matching fingerprints. The right-most point in each graph corresponds to identical chunks or files.

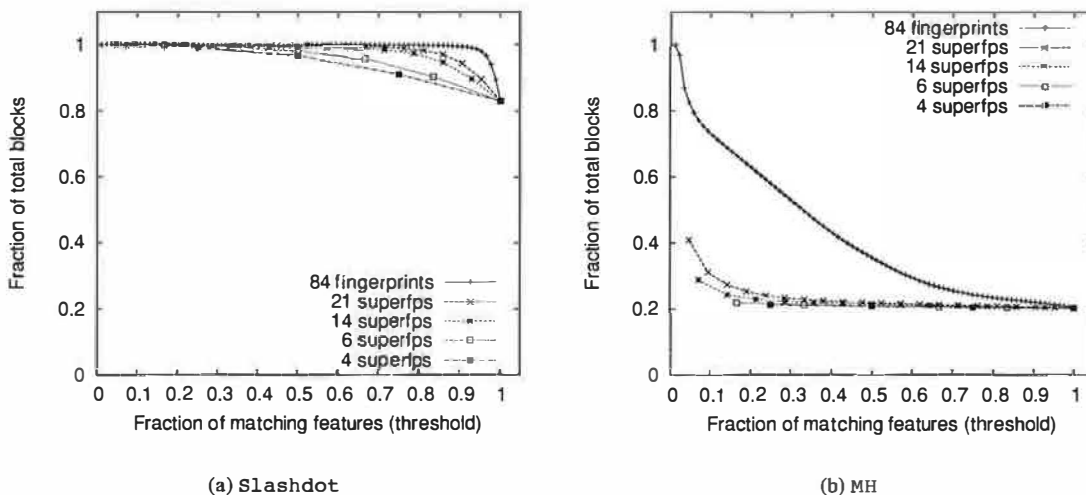


Figure 2: Cumulative distribution of matching fingerprints or super-fingerprints, using 1-KB chunks. The relative shape of the curves demonstrate the much greater similarity in the Slashdot data set than the MH data set.

to a number of super-fingerprints (21 for these two data sets). After this point, *BestFit* encodes the most effectively because taking the first fit with too few fingerprints per cluster is a poor predictor of a match.

Figure 3(b) plots the corresponding execution times. As we increase the number of super-fingerprints, the number of comparisons for *BestFit* to detect similar chunks increases, leading to dramatically greater execution times. The execution times using *FirstFit* are more or less stable and do not show the same sharp in-

crease. In short, using *FirstFit* allows a little space to be sacrificed in exchange for dramatically lower execution times. For example, with Slashdot using *FirstFit*, 1-KB chunks, and six super-fingerprints, REBL produces a relative size of 1.52%; the best *BestFit* number is 1.18% with 42 super-fingerprints. However, the corresponding absolute execution times are 8.1 and 173.5 seconds respectively.

As mentioned in Section 3.1, one interesting parameter that can be modified when using *BestFit* is

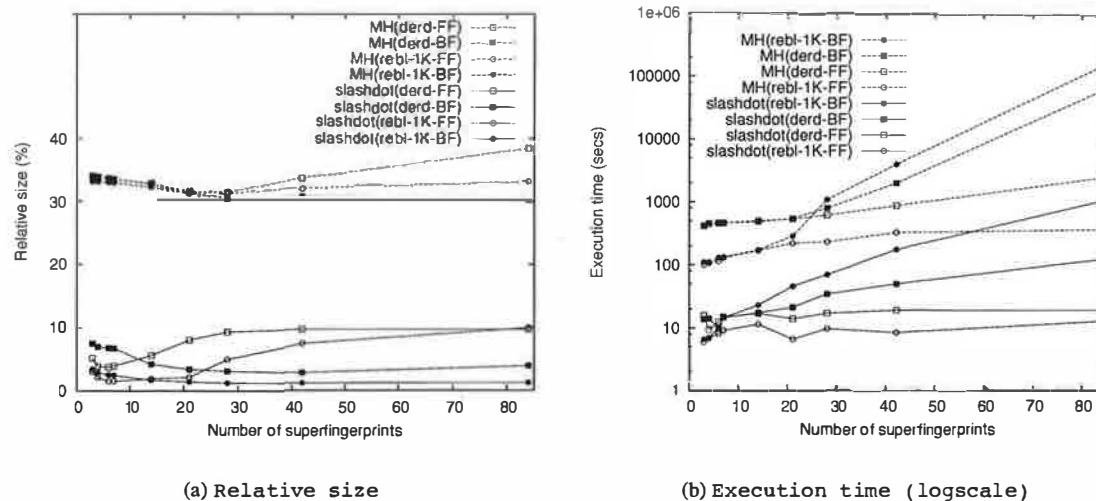


Figure 3: Relative size and total execution time as a function of the number of super-fingerprints, for two data sets, using DERD and REBL.

its eagerness to use the most similar reference chunk against which to delta-encode a version chunk. For instance, one might naturally assume that encoding a chunk against one that matches it in 80/84 fingerprints would be preferable to encoding it against another that matches only 70/84. However, consider a case where chunk A matches chunk B in 82 fingerprints, chunk C in 75, and chunk D in 70; C and D resemble each other in 80/84. Encoding A against B and C against D generates two small deltas and two reference chunks, but encoding B, C, and D against A results in slightly larger deltas but only one unencoded chunk. As a result of this eagerness, *FirstFit* often surprisingly encoded better than *BestFit* until we added an approximation metric to *BestFit*, which lets a given chunk be encoded against a specific reference chunk if the latter chunk is within a factor of the best matching chunk. Empirically, allowing matches within 80-90% of the best match improved overall effectiveness, as shown in the sensitivity analysis in Section 5.3.2.

5.1.4 Benefits of Chunking

While REBL applies super-fingerprints to content-defined chunks, super-fingerprints could also be applied to entire files, similar to detecting commonality in web pages [5]. This would amount to a modification of the DERD approach, optimizing the resemblance detection step, but applying them to entire files can potentially reduce the number of files identified as being similar.

Referring again to Figure 3(a), but this time considering the curves for DERD as well as 1-KB chunks, we see that REBL is always at least as effective as DERD for

both *FirstFit* and *BestFit*. The curves for 4-KB chunks are omitted in order to keep the graphs readable, but generally follow the 1-KB chunk curves with slightly more space consumed.

The corresponding execution times for DERD and REBL are plotted in Figure 3(b). As expected, the greatest execution time is for REBL with *BestFit*; breaking each file into chunks results in more comparisons. Execution times for *BestFit* increase sharply with increasing numbers of super-fingerprints. The best overall results considering both effectiveness and efficiency are with the *FirstFit* variant of REBL using a small number of super-fingerprints. Using 4-KB chunks (not shown) proves to be moderately faster than 1-KB chunks.

One might ask whether it is sufficient to simply use some number (N) of fingerprints rather than combining a larger number of fingerprints into the same number (N) of super-fingerprints. In fact, with *BestFit*, using as few as 14 fingerprints is nearly as effective as using 84 fingerprints. However, even with only 14 fingerprints, the execution cost of *BestFit* is substantially greater than *FirstFit*. Figure 4 reports relative sizes and execution times for the Slashdot data set as a function of the number of fingerprints or super-fingerprints, using an average chunk size of 4 KB. With super-fingerprints and *FirstFit*, relative size increases with more super-fingerprints, while with fingerprints and *BestFit* relative size decreases with more fingerprints. On the other hand, execution time with *BestFit* increases sharply with more fingerprints. The effectiveness with super-fingerprints using *FirstFit* is similar to that using a larger number of fingerprints and *BestFit*.

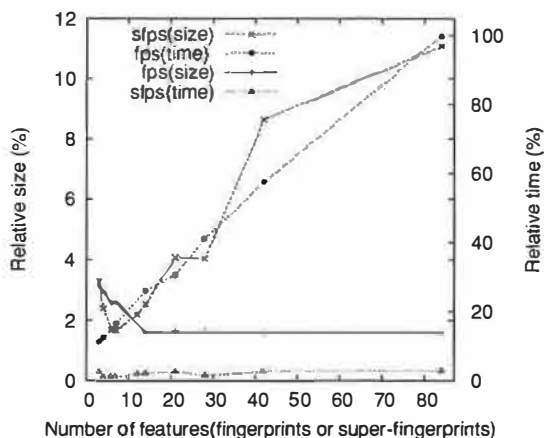


Figure 4: Comparing effect of reduced fingerprints per block and *FirstFit* with super-fingerprints using the Slashdot data set.

To quantify these differences with the example in Figure 4, *FirstFit* with seven super-fingerprints has a relative size of 1.64%; the best effectiveness using fingerprints is 1.57% with 84 fingerprints per chunk. However, the former runs in just 1.4% of the time taken by the latter. The relative execution time with 14 fingerprints and *BestFit*, which gives comparable results to using 84 fingerprints, is 26.0%. Thus, lowering the number of fingerprints per chunk to reduce comparisons (and increase efficiency) may not yield the best encoding size and execution times. In contrast, the use of super-fingerprints and the *FirstFit* variant of REBL is both effective and efficient.

5.2 Comparison of Techniques

In this section, we compare a variety of techniques, focusing on effectiveness and briefly discussing efficiency as indicated by execution times. Table 2 reports sizes compared to the original data set. The relative sizes with CDC are reported for average chunk sizes of 1 KB (with and without WFC) and 4 KB (with WFC). REBL numbers use average chunk sizes of 1 KB (except the 7-GB Users set) and 4 KB, and they include both PBC and WFC.

For the experiments using these data sets, we strove for consistency whenever possible. However, there are some cases where varying a parameter or application made a huge difference. In particular, *gzip* produces output somewhat smaller than *vcdiff* for all our data sets except Slashdot, for which it is nearly an order of magnitude larger. We report the *vcdiff* number in that case only.

For both REBL and DERD, the table gives numbers for 14 super-fingerprints using *FirstFit*. Full compar-

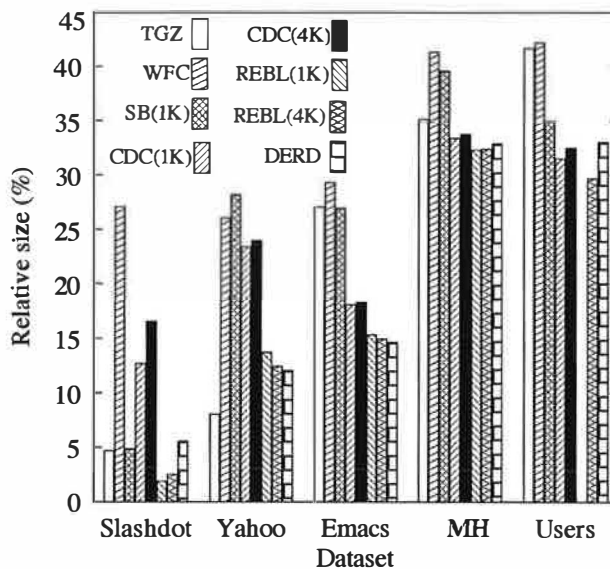


Figure 5: Effectiveness of different encoding techniques based on the relative sizes of the encoded data sets. There is no bar for 1-KB chunks for REBL on the Users data set, as this was not computed. SB refers to SLIDINGBLOCK.

isons of regular fingerprints generally gave a smaller encoding, but at a disproportionately high processing cost as the experiments above demonstrated. REBL had the smallest encoding size in three out of the five data sets above. REBL encoded more effectively than CDC, SLIDINGBLOCK, and WFC with all data sets; it was better than TGZ except with the Yahoo data set; and it was better than DERD except with the Yahoo and Emacs data sets.

Figure 5 graphically depicts the data in Table 2, and Figure 6 shows a scatterplot of how the other techniques compare to REBL. For consistency, we compare the encoding sizes of CDC (1-KB chunks) with REBL (1-KB chunks) and CDC (4-KB chunks) with REBL (4-KB chunks). Users is compared to REBL with 4-KB chunks throughout. Otherwise, REBL with 1-KB chunks is used as the baseline.

As with REBL using 1-KB chunks, REBL using 4-KB chunks is better than TGZ (except with the Yahoo data set), WFC, SLIDINGBLOCK, CDC with either 1-KB and 4-KB chunks and DERD except the Yahoo and Emacs data sets. The effectiveness of REBL compared to TGZ varied by factors of 0.59-2.46, WFC by 1.28-14.25, SLIDINGBLOCK by 1.18-2.56, CDC by 1.03-6.67 and DERD by 0.88-2.91.

Additionally, we compare the effectiveness of REBL with and without using WFC. The relative sizes of the Slashdot, Yahoo, Emacs, and MH data sets using REBL with 1-KB chunks and without WFC are 1.9%, 14.85%, 17.8% and 36.0% respectively. REBL without

Data set	TGZ	WFC	PBC (1 KB)	Sliding Block (1 KB)	CDC			REBL		DERD (14 FF)
					(1 KB w/o WFC)	(1 KB w/ WFC)	(4 KB w/ WFC)	(1 KB 14 FF)	(4 KB 14 FF)	
Slashdot	4.68	27.08	44.46	4.86	12.74	12.68	16.56	1.89	2.52	5.54
Yahoo	8.03	26.02	40.67	28.16	29.18	23.38	23.93	13.72	12.42	12.03
Emacs	27.02	29.27	42.25	26.89	24.95	17.99	18.29	15.31	14.94	14.64
MH	35.11	41.30	48.23	39.57	38.10	33.36	33.73	32.28	32.38	32.87
Users	41.67	42.19	49.93	34.94	34.49	31.48	32.47	N/A	29.69	33.01

Table 2: Data sets and their relative encoding sizes (in percent) as compared to the original size using different encoding techniques. The best encoding for a data set is in **boldface**. TGZ stands for tar plus gzip, WFC is whole-file compression, and PBC is content-defined block-level compression. REBL uses 14 super-fingerprints and *FirstFit*.

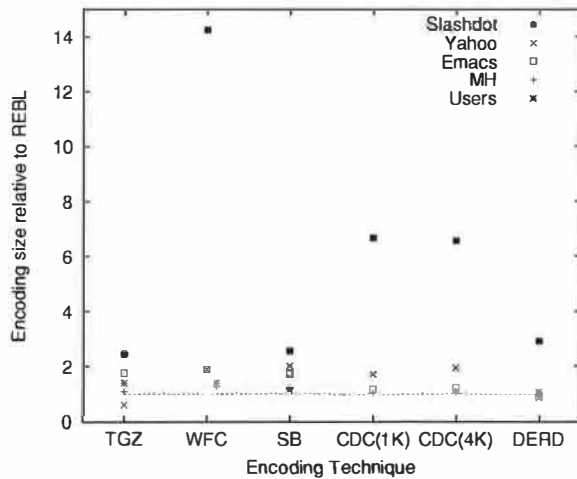


Figure 6: Effectiveness of different encoding techniques, grouped by technique, relative to REBL. REBL uses a 1-KB chunk size, except for Users (4-KB chunks in all comparisons) and CDC with 4-KB chunks (compared head-to-head with 4-KB REBL). The horizontal line indicates the break-even point; all points above this line reflect cases in which REBL is more effective. SB refers to SLIDINGBLOCK.

WFC with 4-KB chunks using the Users data set has a relative size of 30.6%. The corresponding relative sizes of REBL with WFC are reported in Table 2. Without WFC, the relative sizes for two of the data sets (Slashdot and Users) are within 3% of the sizes that include WFC, but REBL without WFC encodes worse than REBL with WFC for the Yahoo, Emacs, MH and data sets by 7.5%, 16.2%, and 11.5% respectively.

An obvious question is how the execution time of REBL compares to the other approaches we have discussed. We have already compared REBL and DERD in some depth. Since REBL relies on CDC, it is inherently more costly than CDC, which in turn requires substantially more computation than a simple technique such as TGZ. How much more costly REBL is depends on how many deltas are computed and how much computation

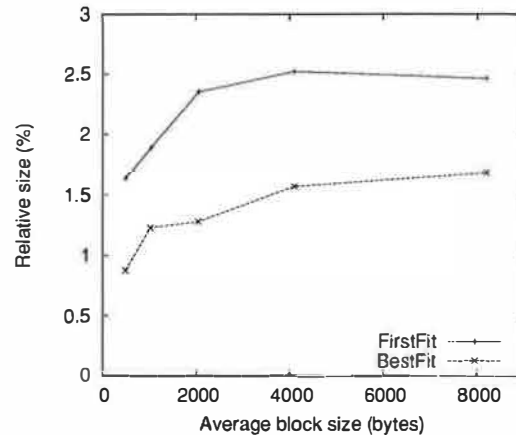


Figure 7: Effect of average block-size on relative size using REBL on the Slashdot data set.

is performed to select chunks to delta. The *FirstFit* variant requires processing that scales linearly, rather than quadratically, with the input size, just as the CDC processing does. Hence the additional cost is comparable to CDC processing. The additional space savings varies across data sets; for Slashdot the additional savings seems easily warranted, while for MH it seems unlikely to be worthwhile.

5.3 Additional Considerations

This subsection describes the sensitivity of REBL to various execution parameters that try to optimize its behavior.

5.3.1 Effect of Average Chunk Size

One potentially important parameter for REBL is the average chunk size. As discussed in Section 5.1.1, smaller chunk sizes provide more opportunity to find similar chunks for delta-encoding. Figure 7 reports results of experiments with the Slashdot data set, 84 fingerprints per chunk, and varying the average chunk sizes from 512 to 8192 bytes. In the case of Slashdot, for

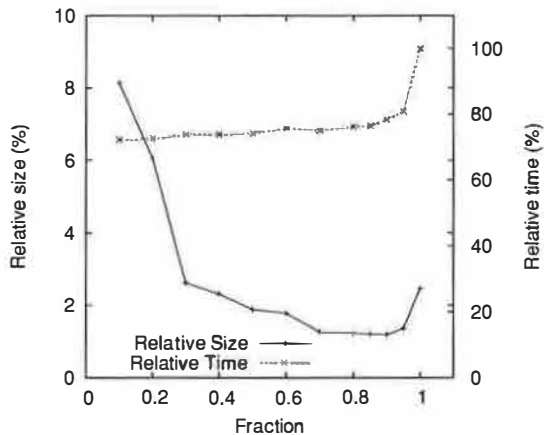


Figure 8: Effect of varying the *BestFit* threshold on relative size using REBL on the Slashdot data set.

both *FirstFit* and *BestFit*, increasing the average chunk size results in larger encoding sizes. The smallest relative size is obtained with the 512-byte chunk size in both variants of REBL.

The same experiment was performed with the MH data set using the *FirstFit* variant of REBL. In this case too, the smallest relative size (32.3%) was obtained using a chunk size of 512 bytes and other chunk sizes reported slightly larger sizes. However, there was minimal degradation in effectiveness moving to larger sizes (the maximum relative size was 32.4% with a chunk size of 8 KB).

Choosing a smaller chunk size provides more opportunities for delta-encoding and better space savings but at a higher run-time cost. For example, for the MH data set, 8-KB chunks save about 20% of the REBL post-CDC processing time, compared to 4-KB chunks, for almost identical encoding effectiveness.

5.3.2 Approximate Matching for *BestFit*

Another parameter we evaluated is the threshold for approximate matching of *BestFit* chunks. Without this threshold, using *BestFit*, a chunk is encoded against another with which it has the most matching fingerprints or super-fingerprints. For a given reference chunk, the “*BestFit* threshold” determines how loose this match can be, permitting the encoding of any chunks within the specified fraction of the best match. Note that in all cases, the system is counting matches and paying the $O(n^2)$ complexity. A very small fraction (low threshold) approximates the *FirstFit* approach in terms of effectiveness, but not efficiency, as it counts the matches but then largely disregards them.

Figure 8 shows the effect of varying the *BestFit* threshold on the Slashdot data set using 84 fingerprints per chunk and an average chunk size of 1 KB. The

graph indicates that as the threshold increases, effectiveness increases, up to about 90%. A threshold of one corresponds to the most precise match, but it actually misses opportunities for delta-encoding, resulting in increased encoding size. A threshold between 0.7 and 0.9 yields the smallest encoding sizes with regular fingerprints for the Slashdot data set; other data sets show similar trends. As expected, the figure also shows increasing relative times as the threshold increases.

5.3.3 Effect of Shingle Size

A shingle specifies the size of a window that slides over the entire file advancing one byte a time, producing a Rabin fingerprint value for each fixed-size set of bytes. The Rabin fingerprints are used to flag content-defined chunk boundaries and to generate features for each chunk that can be used to identify similar ones. We performed experiments varying the shingle size, and using the Slashdot and MH data sets with 84 fingerprints per chunk, 14 super-fingerprints, an average chunk size of 4 KB, and the *FirstFit* variant.

With the Slashdot data set, shingles of four or eight bytes get much less benefit from REBL than larger sizes, but otherwise the analysis is noisy and several disjoint values give similar results. Shingle sizes of 20 and 44 bytes yield similar relative encoding sizes of 2.25% and 2.19% respectively and shingle sizes of 12 and 24 bytes yield relative sizes of 2.52% and 2.53% respectively, whereas a shingle size of 16 bytes results in a relative size of 5.12%. The maximum relative size of 6.84% is obtained with a shingle size of 8 bytes and the minimum of 2.19% with a shingle size of 44 bytes.

In the case of the MH data set, we found that varying the size of a shingle did not vary the relative sizes by as much as in the Slashdot data set, though they did impact processing time. The minimum relative size was 31.2% with a shingle size of 8 bytes and maximum relative size was 32.5% with a shingle size of 44 bytes. However, the 8-byte shingles took 156 CPU-seconds, while 12-byte shingles took 114 CPU-seconds (27% less) to encode the data set to 31.5% of the original (0.7% more).

We conclude that past work that used four-byte shingles [18] may have found their resemblance detection system to be noisier than necessary, but sizes of twelve bytes or more are probably equally arguable.

6 Conclusions and Future Work

In this paper, we introduced a new encoding scheme for large data sets: those that are too large to encode monolithically. REBL uses techniques from compression, duplicate block suppression, delta-encoding, and super-fingerprints for resemblance detection. We have implemented REBL and tested it on a number of data

sets. The effectiveness of REBL compared to TGZ varied by factors of 0.59-2.46, WFC by 1.28-14.25, SLIDINGBLOCK by 1.18-2.56, CDC by 1.03-6.67 and DERD by 0.88-2.91.

We have compared two variants for similarity detection among blocks, *FirstFit* and *BestFit*, and demonstrated that *FirstFit* with super-fingerprints produces a good combination of space reduction and execution overhead. Super-fingerprints are good approximations of regular fingerprints in all the data sets we experimented with. A low threshold of matching super-fingerprints usually results in similar effectiveness to that obtained using a higher threshold for regular fingerprints, but with a dramatically lower execution cost (orders of magnitude in some cases).

The effectiveness of REBL in our experiments is always better than WFC and CDC. However, this is because it incorporates the technology of compression at the file and block level, and the suppression of duplicate blocks, before adding delta-encoding. In fact, compressing individual chunks (or blocks) in any sort of CDC or SLIDINGBLOCK system seems an essential optimization unless the rate of duplication is substantially higher than we have seen in these data sets. This is consistent with the earlier SLIDINGBLOCK work [8], which found that SLIDINGBLOCK needed to incorporate block-level compression to be competitive with *gzip*. Compressing entire files when none of its pieces are suppressed as a duplicate similarly offers benefits.

In some cases, REBL encodes noticeably better than DERD; in the other cases, REBL is very similar to it. The key difference between REBL and DERD comes from dramatic reductions in execution times. The average block size used to mark content-defined blocks affects the encoding sizes of REBL to a limited extent; the more similarity there is in a data set, such as Slashdot, the more effective smaller blocks are.

We are currently working on extending and experimenting with the REBL and LBFS techniques to reduce network usage in communication environments. This will be helpful to determine the applicability of REBL in reducing redundant network traffic, and it can also be compared with other network-oriented mechanisms like *rsync* [23, 29] and link-level fingerprint-based duplicate detection [25]. We would also like to evaluate the effectiveness of these techniques in new environments, such as the Google GMail™ system, which may offer additional opportunities for large amounts of data with subtle variations. Finally, additional detailed comparisons of the wide variety of encoding techniques may offer the opportunity to consider new metrics, such as “bytes saved per cycle,” in selecting among the alternatives.

Acknowledgments

We thank Ramesh Agarwal, Andrei Broder, Windsor Hsu, Arun Iyengar, Raymond Jennings, Leo Luan, Sridhar Rajagopalan, Andrew Tridgell, Phong Vo, Jian Yin, and the anonymous referees for comments and discussions. Special thanks go to our shepherd, Scott Kaplan, for his guidance and understanding. We thank David Mazieres and his research group for making the LBFS code available, Windsor Hsu and Tim Denehy for making the SLIDINGBLOCK code available, and Phong Vo and AT&T for making *vcodex* available.

References

- [1] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. *Journal of the ACM*, 49(3):318–367, May 2002.
- [2] The Internet Archive. <http://www.archive.org/>, 2004.
- [3] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.
- [4] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, 1997.
- [5] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in Lecture Notes in Computer Science, pages 1–10, Montréal, Canada, 2000. Springer-Verlag, Berlin.
- [6] Mun Choon Chan and Thomas Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of Infocom'99*, pages 117–125, 1999.
- [7] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 285–298. USENIX, December 2002.
- [8] Timothy E. Denehy and Windsor W. Hsu. Reliable and efficient storage of reference data. Technical Report RJ10305, IBM Research, October 2003.
- [9] Fred Douglass and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of 2003 USENIX Technical Conference*, June 2003.
- [10] Val Henson. An analysis of compare-by-hash. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.
- [11] David G. Korn and Kiem-Phong Vo. Engineering a differencing and compression data format. In *Proceedings of the 2002 Usenix Conference*. USENIX Association, June 2002.
- [12] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing, Springer Verlag (Heidelberg, FRG and*

- New York NY, USA)-Verlag Surveys, ; ACM CR 8902-0069, 19(3), 1987.
- [13] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, January 1994.
 - [14] J. Mogul, B. Krishnamurthy, F. Douglass, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. *Delta encoding in HTTP*, January 2002. RFC 3229.
 - [15] Jeffrey Mogul, Fred Douglass, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM'97 Conference*, pages 181–194, September 1997.
 - [16] Jeffrey C. Mogul, Yee Man Chan, and Terence Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, March 2004.
 - [17] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.
 - [18] Zan Ouyang, Nasir Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *International Conference on Web Information Systems Engineering (WISE)*, December 2002.
 - [19] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the 2004 Usenix Conference*, June 2004.
 - [20] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, Monterey, CA, 2002.
 - [21] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
 - [22] Sean C. Rhea, Kevin Liang, and Eric Brewer. Value-based web caching. In *Proceedings of the twelfth international conference on World Wide Web*, pages 619–628. ACM Press, 2003.
 - [23] rsync. <http://rsync.samba.org>, 2004.
 - [24] rzip. <http://rzip.samba.org/>, 2004.
 - [25] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, August 2000.
 - [26] Federal Information Processing Standards. Secure hash standard. FIPS PUB 180-1, April 1995.
 - [27] Torsten Suel, Patrick Noel, and Dimitre Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Proceedings of ICDE 2004*, March 2004. To appear.
 - [28] W. Tichy. RCS: a system for version control. *Software—Practice & Experience*, 15(7):637–654, July 1985.
 - [29] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.
 - [30] Andrew Tridgell. Personal communication, December 2003.

Notes

¹In practice we do not expect the complexity to be this bad, and some heuristics could be used to reduce it [30], but they are beyond the scope of this paper. Even with such optimizations, the techniques described in this paper improve efficiency substantially.

²Encoding chains are possible— A against B , B against C , and so on—but decoding such a chain requires first computing B from C to obtain A . We discount this possibility due to its complexity and performance implications.

³Our initial implementation stored the relationship of every pair of blocks with at least one matching fingerprint. With this approach, we ran out of address space operating on our larger data sets. We reduced memory usage by storing a information only for blocks matching many fingerprints (a default of $\frac{1}{4}$), but even that approach suffers on extremely large data sets.

⁴Other sophisticated techniques may be used to generate super-fingerprints, but in our case, we needed a hashing function with a low probability of collisions and MD5 satisfied the criteria.

Alternatives for Detecting Redundancy in Storage Systems Data

Calicrates Policroniades and Ian Pratt

Computer Laboratory
Cambridge University
Cambridge, UK, CB3 0FD
{name.surname}@cl.cam.ac.uk

Abstract

Storage systems frequently maintain identical copies of data. Identifying such data can assist in the design of solutions in which data storage, transmission, and management are optimised. In this paper we evaluate three methods used to discover identical portions of data: whole file content hashing, fixed size blocking, and a chunking strategy that uses Rabin fingerprints to delimit content-defined data chunks. We assess how effective each of these strategies is in finding identical sections of data. In our experiments, we analysed diverse data sets from a variety of different types of storage systems including a mirrored section of sunsite.org.uk, different data profiles in the file system infrastructure of the Cambridge University Computer Laboratory, source code distribution trees, compressed data, and packed files. We report our experimental results and present a comparative analysis of these techniques. This study also shows how levels of similarity differ between data sets and file types. Finally, we discuss the advantages and disadvantages in the application of these methods in the light of our experimental results.

1 Introduction

Computer systems frequently store and manipulate several copies of the same data. Some applications may generate versions of a document stored as separate files, but whose content differs only slightly. Software development teams, file synchronisers [1, 28, 29], backup systems [19], reference data managers [9], and peer to peer systems [8, 11, 17, 26] deal with large quantities of identical data. Efficient data management solutions may be created if system designers are aware of the amount of redundant data seen in diverse data sets.

Although saving disk space can be useful, over the past few years there has been a constant reduction in the cost of raw disk storage. Some may argue that the disk

space savings obtained by suppressing identical portions of data are of minimal significance. However, apart from the benefits obtained from disk block sharing, there are other factors that need to be considered:

- Storage systems may exploit data duplication patterns to optimise the use of storage space and bandwidth. Single Instance Storage (SIS) [3] explores the content of *whole files* to implement links with semantics of copies instead of storing a file with the same content several times. Backup systems such as Venti [19] store duplicated copies of *fixed size* data blocks only once. LBFS [18], Pasta [16, 17], Pastiche [8], and the Value-Based Web Caching algorithm (VBWC) [23] find identical portions of data using Rabin fingerprints. In this method, data is divided into *content-defined* chunks in order to exploit cross-file data duplication. Additional details of the content-defined chunking method will be presented in the section 3 of this paper.
- File systems may obtain improved caching performance if they are aware of contents shared between files. In this way, it would be possible to provide better hit ratios for a given cache size. A potential size reduction of the main memory file cache may have important performance effects.
- In mobile environments, devices are often *limited in storage and bandwidth*. Furthermore, there are factors such as *energy consumption and network costs* associated with data transmission that can become critical [2, 18]. Under certain circumstances, it might be desirable to perform significant computation to reduce the number of bits transmitted over low-bandwidth or congested links.

In summary, three different methods are frequently used to eliminate duplicated data among files: whole file content hashing, fixed size blocking, and a chunking

strategy that uses Rabin fingerprints to delimit content-defined portions of data. Due to the lack of a practical comparative study, the typical performance of each method and their suitability for different data profiles are not clear. In this work, we evaluate the effectiveness of these three methods to discover data redundancy and show the potential benefits of their employment under diverse data profiles. We have developed a set of programs to evaluate each of the approaches, analyse data redundancy patterns, and expose practical trade offs. We explored different collections of real-world data sets to determine how sensitive these methods are to different data profiles:

- **Mirrored section of sunsite.org.uk**¹. This data is a subset of an Internet archive and its size is over 35 GB. Compressed and packed data were common in this data set.
- **Users' personal files.** The data analysed in this collection of files is held in 44 home directories of different users in the Cambridge University Computer Laboratory. The size of this data set is approximately 2.9 GB.
- **Research groups' files.** This data set contains collections of files associated with different research projects of the Computer Laboratory. This is a data set with a potentially high level of data duplication because it stores software development projects, shared documents, and information accessed and manipulated by groups of people. The size of this data set was 21 GB.
- **Scratch directories.** The 100 GB of information explored in this section represents the largest and most diverse data set analysed. We thought that this collection of files is a good example of a data set where no obvious interrelationship is previously known.
- **Software distributions.** To explore the sharing patterns of highly correlated data in different states, we explored five successive Linux kernel distributions in three different formats: packed and compressed (.tar.gz), uncompressed but still tarred (.tar), and uncompressed and untarred.

We had a special interest in the method that uses Rabin fingerprints to delimit chunks of identical data because it has been used in Pasta [16, 17], an experimental large-scale peer to peer file system developed at the Computer Laboratory. We have plans to incorporate an optimised version of Pasta into the XenoServers [22] project. This study enabled us to assess the advantages and drawbacks of using the content-defined data chunking strategy as

a compression and replica management tool in the next implementations of the file system. The results directly helped us to understand the impact of these techniques in Pasta, and we believe our experimental results may also be useful to other parties.

The next section of this paper presents an overview of the related work. In section 3 we introduce the reader with the methodology used to measure the levels of identical data in large collections of files. Our experiments and results are presented in section 4. Finally, in section 5 we conclude with a discussion of our findings.

2 Related Work

A number of strategies to discover similar data in files have been explored in different systems. Unix tools such as `diff` and `patch` can be used to find differences between two files and to transform one file into the other. Rsync [28] copies a directory tree over the network into another directory tree containing similar files. It saves bandwidth by finding similarities between files that are stored under the same name.

The Rabin fingerprinting algorithm [20] has been employed with different purposes such as fingerprinting of binary trees and directed acyclic graphs [4, 13], or as a tool to discover repetitions in strings [21]. However, we are interested in how Rabin fingerprints can be used to identify identical portions of data in storage systems. In general, Rabin fingerprints have been used for this purpose in two ways: to sample files in order to discover near-duplicate documents in a large collection of files, or to create content-defined chunks of identical data.

Manber [15] employs Rabin fingerprints to sample data in order to find similar files. His technique computes fingerprints of all possible substrings of a certain length in a file and chooses a subset of these fingerprints based on their values; the selected fingerprints provide a compact representation of a file that is then used to compare against other fingerprinted files. Similarly, Broder applies resemblance detection [5] to web pages [6] in order to identify and filter near-duplicate documents. Rabin fingerprints of a sliding window are computed to efficiently create a vector of shingles of a given web page. Consequently, instead of comparing entire documents, shingle vectors are used to measure the resemblance of documents in a large collection of web pages. The techniques used by Manber and Broder have been adapted by Spring and Wetherall [27] to eliminate redundant network traffic. However, they used Rabin fingerprints as pointers into data streams to find regions of overlapping content before and after the fingerprinted regions.

Different systems use blocking strategies that employ the Rabin fingerprinting algorithm to create *content-defined* and *variable-sized* data chunks. Probably the first

storage system that used Rabin fingerprints for this purpose was LBFS [18], specially designed to transmit data over low-bandwidth networks. Using the Rabin fingerprinting algorithm, LBFS finds similarities between files or versions of the same file. It avoids retransmission of identical chunks of data by using valid data chunks contained in the client's cache and by transmitting to and from the data server only the chunks that have been modified. LBFS's chunking algorithm was tested on a data set of 354 MB reporting that around 20% of the data was contained in shared chunks.

Blocking in Pasta [16, 17], an experimental peer to peer file system, also exploits the benefits of common information between files. Caching and replica placement are defined by data blocks' content. These blocks are built by computing Rabin fingerprints of the file data over a sliding window. Identical blocks are stored only once and referenced using a shared key. A similar technique is used in Pastiche [8]. In Value-Based Web Caching [23], web proxies index data according to their content and avoid retransmission of redundant data to clients connected over low-bandwidth links. Although all these systems show that improved block sharing levels can be obtained using content-defined chunking strategies, they do not present broad experimental results based on diverse data sets.

Data redundancy in storage systems has also been identified using *fixed size blocking strategies*. Sapuntzakis et al., aimed to reduce the amount of data sent over the network by identifying identical portions of data in memory [25]. They use a hash-based compression strategy of memory aligned pages (i.e. fixed size blocks of data) to accelerate data transfer over low-bandwidth links and improve memory performance.

Venti [19], a network storage system intended for archival of data, aims to reduce the consumption of storage space. It stores duplicated copies of fixed size data blocks only once. Venti reports a reduction of around 30% in the size of the data sets employing this method. Future implementations of Venti may also incorporate a content-based blocking scheme based on Rabin fingerprints.

A different approach to eliminate data redundancy can be seen in SIS [3] for Windows 2000. It saves space on disk and in main memory cache but with a different approach; SIS explores the content of the *whole file* and implements links with the semantics of copies for identical files stored on a Windows 2000 NTFS volume. A user level service, called the *groveler*, is responsible for automatically finding identical files, tracking changes to the file system, and maintaining a database with the corresponding file indexes. When SIS was tested on a server with 20 different images of Windows NT the overall space saving was 58%.

Although many systems have proposed different techniques to manage duplicated data, it has been only lately that practical studies to assess their benefits and applicability have been performed. Building on Manber's observations, Douglass and Iyengar [10] explore duplication in empirical data sets using Delta-Encoding via Resemblance Detection (DERD) and quantify their potential benefits. Their technique generalises the applicability of delta-encoding by choosing an appropriate set of base versions in a large collection of files through resemblance detection.

As part of the design of a storage system specially crafted to manage reference data [9], a comparison on the effectiveness of three duplicate suppression techniques has been done; two of the techniques analysed in this work are similar to the methods we analysed: fixed size blocking and the content-defined chunking algorithm. The third technique, called *sliding blocking*, uses rsync checksums and a block-sized sliding window to calculate the checksum of every overlapping block-sized segment of a file. The sliding blocking technique consistently detected greater amounts of redundant data than the other two strategies.

More recently, Redundancy Elimination at the Block Level (REBL) has been proposed as an efficient and scalable mechanism to suppress duplicated blocks in large collections of files [14]. REBL combines advantageous features of techniques such as content-defined data chunks, compression, delta-encoding, and resemblance detection. Empirical data sets were used to compare REBL with other techniques. REBL presented the smallest encoding size in 3 out of the 5 data sets analysed and consistently performed better than the other techniques. Specifically, the effectiveness of this technique compared to the content-defined chunking strategy varied by factors of 1.03-6.67, whole file compression by 1.28-14.25, sliding blocks [14] by 1.18-2.56, object compression (tar.gz) by 0.59-2.46, and DERD by 0.88-2.91. All these studies [9, 10, 14] used Rabin fingerprints as a tool to eliminate data redundancy. Thus, their experimental results relate highly to ours.

3 Design

In this section we explain the methodology used to quantify duplication in our data sets. In general, we analyse a collection of files and spot identical data among them using the three methods mentioned before: whole file content, fixed size blocks, and Rabin fingerprints. As a result, we collect information to exhibit sharing patterns in the different data sets.

Sharing patterns at a **whole file** granularity are found by calculating the SHA-1 digest of individual files. The first 64 bits of the resulting digest are used as the key to

a hash table. Identical files are likely to be indexed using the same hash table entry. We use the hash table to store statistical data of identical files such as number of occurrences and size of the original file. Although Henson [12] warns about some of the dangers of comparing by hash digests, this method can be safely used in our experiments. In our case, false sharing of the key space is not a crucial concern².

In order to find similarities using **fixed size blocks** an analogous procedure is employed, but instead of obtaining digests for whole files, they are calculated for contiguous non-overlapping fixed size portions of the files. In this case, hash table entries correspond to unique data blocks.

The third method analysed employs Rabin fingerprints. It offers the advantage that the chunks generated are **defined according to their contents**. The mathematical principles of Rabin fingerprints are well documented [20]. A Rabin fingerprint $f(A)$ is the polynomial representation of some data $A(t)$ modulus an irreducible polynomial $P(t)$. We compute such an irreducible polynomial only once and use the same value in all our experiments in order to find identical pieces of data. The algorithm for computing such a polynomial can be found in [7]. Our implementation follows the principles presented by Broder in [4] which is an extension of the work done by Rabin. Broder uses precomputed tables to process more than 1 bit at a time, particularly, 32 bits are divided into four bytes and processed in one iteration. The value of k , which is the degree of the irreducible polynomial and consequently the length of the fingerprint, should be a multiple of 32.

To divide a file into content-defined chunks of data our implementation incrementally analyses a given file using a sliding window and marks boundaries according to the Rabin fingerprints obtained in this process. Figure 1 depicts this mechanism. The program inspects every w bytes of a sliding window that is shifted over the contents of the file. Although the value of w can be tuned, changing the size of the sliding window does not significantly impact the result. Experimental evidence in [16, 18] shows that by setting $w = 48$ bytes it is possible to discover significant levels of data duplication.

Adding a new byte into the sliding window is accomplished in two parts. Firstly, the value for the oldest byte in the window are subtracted from the fingerprint. Secondly, the terms of the new byte are added to the fingerprint. This is possible since the fingerprint is distributive over addition. When subtracting, precomputed tables are used in order to improve the implementation performance.

Rabin fingerprints for each window frame are calculated and if the value obtained matches the r least significant bits of a constant, a breakpoint is marked. These

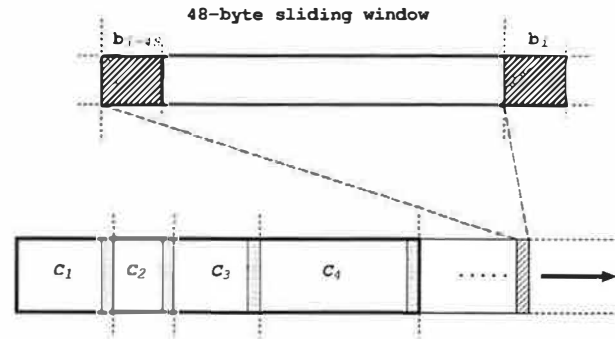


Figure 1: Chunks' boundaries are found using a 48-byte sliding window that incrementally analyses the file's content and computes Rabin fingerprints. Shaded boxes represent the 48-byte regions that generated a boundary. The light striped rectangle corresponds to the current 48-byte window. At each step the byte in the oldest position of the sliding window (b_{i-48}) is subtracted from the fingerprint and the next byte in the file (b_i) is added to the fingerprint.

breakpoints are used to indicate chunk boundaries. In order to avoid pathological cases (i.e many small blocks or enormous blocks) our implementation forces a minimum and a maximum block size.

Once a boundary has been set, the SHA-1 digest corresponding to the chunk's content is calculated. Similar to the other two techniques, the first 64 bits of the SHA-1 digest are used as the key for accessing the hash table that stores statistical information about identical data chunks.

Two kind of values are calculated and reported in our experimental results: percentage of identical data and storage savings. To calculate the **percentage of identical data** in shared blocks we add for each duplicated block, the product of its size and its number of occurrences; then, present this value as a percentage of the original data set size.

To calculate **storage space savings** we add the size of every unique block in the hash table; replicated blocks are counted only once. We present this value as a percentage of the original data set size. Additionally, storage space savings are compared with the space used by simply **tarring and compressing** the whole collection of files in the different data sets. We used the standard `tar` and `gzip` utilities for this purpose.

4 Repeated Data in Empirical Data Sets

This section presents the experimental results used to evaluate the benefits and performance of three different duplicate suppression methods in each of the data sets mentioned.

4.1 Mirror of sunsite.org.uk

In order to find commonality in data that resembles a standard Internet archive, we ran our programs on a 35 GB section of sunsite.org.uk. The total number of files in the data set was 79,551, with an average file size of 464 KB. Table 1 shows a partial characterisation of this data set. It shows the 15 most popular file-name extensions and their percentage of the total number of files. The 15 most popular file extensions account for over 82% of all files. Table 1 also indicates the 15 file extensions that use the most storage space and the percentage of the total space they consume; collectively, they cover almost 97% of the whole data set. Packed and compressed files (e.g., rpm, gz, bz2, zip, and Z) represent an important part of the data set (around 24.4 GB). A detailed analysis of compressed files is presented in section 4.1.1.

Rank	Popularity		Storage Space	
	Ext.	% Occur.	Ext.	% Storage
1	.gz	32.50	.rpm	29.30
2	.rpm	10.60	.gz	20.95
3	.jpg	7.54	.iso	20.40
4	.html	4.83	.bz2	6.26
5	.gif	4.43	.tbz2	5.65
6	—	4.16	.raw	4.44
7	.lsm	3.74	.tgz	2.66
8	.tgz	2.90	.zip	2.53
9	.tbz2	2.35	.bin	2.00
10	.Z	2.12	.jpg	0.94
11	.asc	1.84	.Z	0.65
12	.zip	1.59	.gif	0.43
13	.rdf	1.39	.tif	0.31
14	.htm	1.21	.img	0.21
15	.o	1.06	.au	0.19
Total	—	82.26	—	96.92

Table 1: Data profile of our mirrored section of sunsite.org.uk. Files without extension are denoted by the — symbol.

We ran our implementation of the content-defined chunking method over the data using different expected chunk sizes and sliding window lengths. Although in all the subsequent experiments the maximum chunk size permitted was set to 64 KB, the minimum chunk size was fixed to 1/4 of the expected chunk size; the expected chunk size is set as a parameter in the algorithm. By fixing the maximum chunk size to 64 KB despite changes in the expected chunk size, we aimed to maximise the opportunities of finding identical portions of data in larger chunks. We enforce minimum and maximum chunk lengths to avoid pathological cases such as very small or large chunks. Table 2 shows the amount of information in shared chunks with two different window

sizes and three different expected chunk sizes. The last column of Table 2 illustrates the percentage of identical data that was found using fixed size blocks.

Table 2 suggests that the size of the window does not significantly influence the commonality levels. Other studies [16, 18], which use similar techniques to ours, report similar findings. Therefore, in all subsequent experiments we fixed the window size to 48 bytes. We also observe that when the expected size of the chunks is shorter, the percentage of shared data is larger because the probability of finding similar chunks also increases. However, the expected size of the blocks represents a trade-off between the size of the hash table needed to maintain a larger number of entries for each of the unique chunks and the potential storage space saved (see section 4.6).

The percentage of identical data in whole files was only 5%. Therefore, it was possible to find a considerable amount of similar data in partially modified files. On the other hand, the percentages of identical data found using the content-defined chunking method presented only slight differences when compared with those obtained using fixed size blocks. These findings suggest that a storage system handling this kind of data could easily select a fixed size blocking scheme without losing significant storage space savings.

Size	% of Data in Identical Blocks		
	48 B window	24 B window	Fixed Size
8 KB	16.43	16.12	12.13
4 KB	20.27	19.72	17.25
2 KB	25.00	24.18	22.23

Table 2: Percentages of identical data in a 35 GB section of sunsite.org.uk.

Figure 2 shows the distribution of block sizes under the content-defined chunking method when the expected chunk size was set to 8 KB. In particular, we obtained an average block size of 9.2 KB. The algorithm generated 3,730,576 blocks of which 7.6% have at least one identical copy. It is also possible to appreciate the impact of pathological cases on the distribution: the chunks under the minimum block size (2 KB) correspond to files that are shorter than the minimum permitted or to final portions of files. Moreover, the peak at 64 KB corresponds to chunks inserted because of the maximum size allowed.

We focused our attention on the set of files that account for 97% of the total storage space. We explored the levels of similarity of these 15 kinds of files. We fixed the expected chunk size to a value of 4 KB. Table 3 shows the results and compares them against the percentage that can be obtained if we consider whole file contents. We found remarkable patterns in our results. Firstly, it is very difficult to exploit similarity in com-

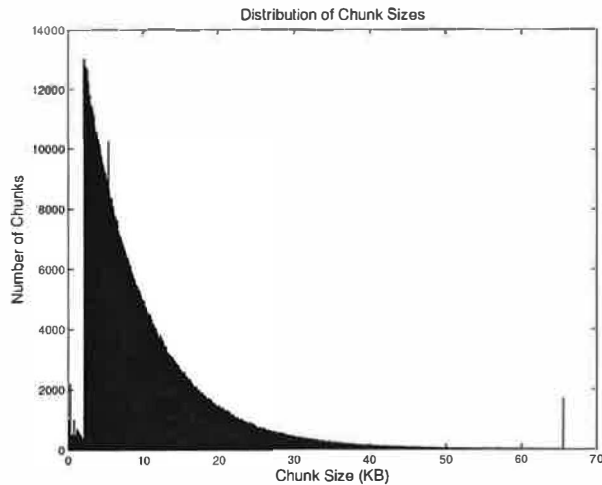


Figure 2: Distribution of chunk sizes obtained from sun-site.org.uk using an expected chunk size of 8 KB and a 48-byte sliding window size.

pressed files; practically all the duplicated chunks were contained in identical files. The behaviour of compressed data will be further investigated in the section 4.1.1.

Secondly, .iso and .img files presented the highest difference in percentage of similarity against the whole file column. These findings suggest that variations between files can be efficiently isolated using the content-defined chunking method, whereas under the whole file approach, even a small change in the file leads to storing a new almost-identical file. The negative effects of this situation are intensified in large files such as ISO image files in which the average size was 280 MB. All other files showed only slight increments if they are compared against the value obtained for the whole file scheme.

Keeping only one copy of the information saves storage space. Using the best scenario, which was the content-defined chunking method with an expected chunk size of 2 KB, our experimental results indicate that a file system would store only 30 GB of unique data instead of the original 35 GB, representing around 14% of storage savings. Duplicate suppression proved to be somewhat more efficient in saving storage space than the tar-compressed version of the data set; the tar.gz file for this data set claimed around 33.3 GB of disk space.

We consider that an explanation to these numbers may be found in the detailed analysis of the files that conform this data set. As has been pointed out before, a large amount of data is already compressed (see table 1); approximately 24.4 GB correspond to rpm, gzip, bz2, zip, and Z files. Note that archives in rpm files are compressed using gzip's deflation method. To a certain extent, redundant data in these files has already been re-

Format	% of Identical Data	
	Content-defined chunks	Whole file content
.rpm	9.08	7.07
.gz	6.71	5.29
.iso	31.26	0.54
.bz2	8.33	8.32
.tbz2	5.02	5.02
.raw	0.47	0.0
.tgz	13.55	13.55
.zip	2.79	1.43
.bin	2.57	0.0
.jpg	0.49	0.28
.Z	3.40	3.14
.gif	2.73	2.69
.tif	95.29	95.29
.img	33.84	8.69
.au	0.0	0.0
all ext.	20.27	5.03

Table 3: Detailed similarity pattern in our mirror of sun-site.org.uk. A 4 KB expected chunk size and a 48-byte sliding window size were used in the content-defined chunking method. The last row of the table shows the values obtained when all files in the data set were analysed.

moved as part of the LZ77 [31] compression technique. Compressing compressed data with the same algorithm normally results in more data, not less.

However, it may still be possible to argue that if the content-defined chunking method was able to remove duplicated chunks of data from the data set, the compressed version of the data would do it as well, resulting in a smaller tar.gz file. The gzip compression algorithm replaces repeated strings in a 32 KB sliding window with a pointer of the form (distance, length) to the previous and nearest identical string in the window. Distances are limited to the size of the sliding window (i.e. 32 KB) and lengths are limited to 258 bytes. As a consequence, redundancy elimination occurs within a relatively local scope; identical portions of data across files will be detected only if files are positioned close in the tar file. In contrast, the content-defined chunking method is able to find data redundancy across distant files in the data set and, in this particular case, to save more storage space than the compressed tar file.

4.1.1 Compressed Data

Compressed files (e.g., gz, bz2, zip, and Z) constitute an important segment of information within our sunsite data set: around 14 GB correspond to compressed files. In this experiment we measured the potential storage space savings that might be obtained if once data is decompressed, the content-defined chunking strategy is used

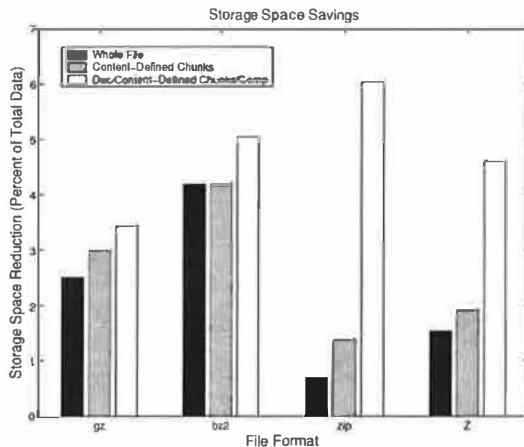


Figure 3: Storage space reductions using three different methods to eliminate duplication in compressed data. Original sizes of the data sets: gz=8.13 GB, bz2=2.4 GB, zip=1 GB, and Z=260 MB.

to suppress duplicates, and then compression is applied again on the resulting set of unique chunks. We decided to use the content-defined chunking method to eliminate data redundancy because it proved to be the most efficient strategy to find data similarity over the four main categories of compressed files. Tools such as `zcat`, `bzcat`, and `unzip` were used to decompress the files. The output stream generated by all these utilities was set as the input to our redundancy elimination program that used a 4 KB expected chunk size.

Figure 3 compares the storage space savings obtained in each of the four categories of compressed files using three different methods to eliminate duplication. The first method suppress duplication from the original collection of compressed files using the whole-file approach. The second method removes similar chunks of data from the original compressed files using the content-defined chunking method. Finally, in the third strategy the files are decompressed, redundancy is removed using the content-defined chunking method, and the resulting unique chunks are compressed again. Although the storage space savings are maximised using the third method, it barely outperforms the result obtained with the content-defined chunking strategy on the original files especially in the cases in which the original data set is of considerable size (gz and bz2 formats).

When duplication is removed from the uncompressed version of the files, the value obtained for the zip category is substantially different to those seen in the other two techniques. It also contrasts with the pattern observed in the other three data sets in which the differences between columns are fairly small. It seems that re-

dundancy elimination specially helped zip files. In general, zip is used to deflate one file at a time to then include it into a single object; it limits any potential size reduction to intra-file compression. In contrast, the other compression tools also remove inter-file data duplication (e.g., from all the files in a tar) which finally reduces the benefits of redundancy suppression due to the content-defined chunking method. We conclude that the improvement seen in zip files can be attributed to the ability of the content-defined chunking method to eliminate redundancy within a broader scope (i.e. inter-file redundancy); a gap that zip compression fails to address.

It seems that the comparatively slight storage savings obtained by decompressing files to eliminate redundancy, and compressing the result anew may not be enough to justify the computational overhead of the whole process. Douglass and Iyengar also analyse commonality patterns in compressed data [10]; they reach a similar conclusion to ours.

4.2 Users' Personal Files

The data analysed in this section was held in 44 home directories of different users of the Computer Laboratory. Although the total amount of data processed was only 2.9 GB, this data set presented high diversity in the kind of files stored; we collected 1,756 different file-name extensions in 98,678 files with an average file size of 31 KB. However, the profile of the data set follows a clear pattern. Table 4 shows the most common file-name extension in terms of popularity and storage space. Apart from the files without extension, home directories are mainly used to store files related to word processing and source code development. The 15 file-name extensions showed in Table 4 under the column related to storage space account for over 69% of the whole data set.

In this case the percentage of identical data in whole files was 12.80%. Table 5 shows the result of running our implementation of the content-defined chunking algorithm over the data set using different expected chunk sizes. It also shows the percentage of identical data found when we explored the files using only fixed size blocks. Although the performance of the content-defined chunking algorithm was better, significant storage space savings can also be obtained by using fixed size blocks.

Our results indicate that using the content-defined chunking method and an expected chunk size of 2 KB, which is the best case, a storage utility would maintain only 2.3 GB of the original 2.9 GB. This represents a storage space reduction of 20.6%. However, the compressed tar version of the data set used only 1.4 GB of disk space; considerably outperforming our best duplicate suppression scenario. Similar storage saving ratios were obtained in the next two data sets (research groups'

Rank	Popularity		Storage Space	
	Ext.	% Occur.	Ext.	% Storage
1	—	19.47	.ps	17.24
2	.eps	4.83	—	11.73
3	.obj	3.98	.gz	10.66
4	.tex	3.82	.pdf	6.16
5	.c	3.43	.eps	4.58
6	.gz	2.85	.zip	4.13
7	.gif	2.45	.doc	3.13
8	.ps	2.28	.ppt	2.60
9	.dat	2.11	.obj	1.75
10	.html	1.81	.xls	1.53
11	.h	1.68	.tgz	1.29
12	.log	1.61	.tex	1.25
13	.aux	1.30	.c	1.24
14	.java	1.28	.so	1.19
15	.dvi	1.26	.txt	1.04
Total	—	54.16	—	69.52

Table 4: Profile of the data in 44 home directories of the Cambridge University Computer Laboratory. Files without extension are denoted by the — symbol.

Size	% of Data in Identical Blocks	
	Content-defined chunks	Fixed size blocks
8 KB	24.16	17.22
4 KB	26.76	18.05
2 KB	29.30	19.25

Table 5: Percentages of identical data obtained in 44 home directories of users of the Cambridge University Computer Laboratory. A 48-byte sliding window size was used in the content-defined chunking method.

files and scratch directories) when their corresponding compressed tar files were generated.

4.3 Research Groups' Files

This data set represents a collection of files stored by work groups. We analysed the information of different research groups in the Computer Laboratory. It presents high degrees of similarity because it contains software projects, documents, and information shared among groups of people. This is an ideal environment to save storage space or to reduce the amount of data transmitted based on suppressing identical portions of data. The data set contained a total of 708,536 files in 21 GB of disk space and a 32 KB average file size. Table 6 illustrates the main sections of information arranged by file-name extension popularity and storage space used. Although we found 2,820 different file extensions, our list with the 15 most popular extensions covers more

Rank	Popularity		Storage Space	
	Ext.	% Occur.	Ext.	% Storage
1	.c	15.82	—	15.56
2	.h	14.51	.gz	10.20
3	—	13.90	.ps	8.01
4	.html	4.07	.c	6.66
5	.o	3.45	.a	3.29
6	.c,v	2.79	.pdf	3.15
7	.h,v	2.11	.o	2.97
8	.py	1.95	.eps	2.41
9	.gif	1.56	.tgz	2.18
10	.S	1.40	.h	1.93
11	.gz	1.23	.o	1.82
12	.if	1.14	.html	1.40
13	.m	1.01	.taz	1.29
14	.eps	0.97	.5	1.24
15	.s	0.85	.tar	1.23
Total	—	66.76	—	63.34

Table 6: Profile of the data stored by different research groups of the Cambridge University Computer Laboratory. Files without extension are denoted by the — symbol.

than 66% of the files. Moreover, the percentage of data within the 15 extensions that use the most storage space accounts for over 63% of the total size of the data set.

The percentage of identical data in whole files was 25%. It clearly demonstrates an increment over the previous data sets. Table 7 shows the percentages of data in shared blocks for the other two methods: content-defined chunks and fixed size blocks for different expected block sizes. The rates of commonality obtained under the fixed size approach also present high levels of commonality although they are substantially behind the content-defined chunks' percentages.

Under this ideal scenario, not only due to the high amount of identical data contained in whole files but also due to the potential relationships between the files analysed, the use of Rabin fingerprints proved its efficiency. Our results indicate that using an expected chunk size of 2 KB, a storage system would hold only 14 GB in unique blocks in contrast with the 21 GB of the original data set. This means a reduction in storage space of around 33%. Once more, the compressed collection of files (i.e. tar.gz format) used less storage space than the duplicate suppression techniques; it claimed only 8.9 GB of disk space.

4.4 Data Stored in Scratch Directories

The 100 GB of data explored in this section represents the largest data set studied. It contained a total of 1,959,883 files with an average file size of 55 KB. Ta-

Size	% of Data in Identical Blocks	
	Content-defined chunks	Fixed size blocks
8 KB	37.01	28.77
4 KB	39.61	29.62
2 KB	44.59	32.94

Table 7: Percentages of identical data found in several research groups' directories of the Cambridge University Computer Laboratory. A 48-byte sliding window size was used in the content-defined chunking method.

Rank	Popularity		Storage Space	
	Ext.	% Occur.	Ext.	% Storage
1	.c	16.47	.log	33.41
2	.h	15.05	—	24.04
3	—	13.14	.gz	2.76
4	.o	6.55	.c	1.52
5	.0	4.00	.txt	1.46
6	.d	3.99	.o	1.24
7	.gz	1.94	.ps	0.81
8	.3	1.89	.a	0.70
9	.S	1.47	.pdf	0.69
10	.py	1.26	.ul2	0.60
11	.s	1.25	.xls	0.57
12	.ih	1.09	.dl1	0.56
13	.al	0.92	.ill	0.55
14	.l	0.84	.prof	0.55
15	.ast	0.81	.frag	0.53
Total	—	70.67	—	69.99

Table 8: Profile of the data stored in scratch directories in machines of the Cambridge University Computer Laboratory. Files without extension are denoted by the — symbol.

ble 8 gives a partial characterisation of the data set. Once more, it presents the information organised in two main columns according to file-name extension popularity and storage space used. This time the top 15 files, in terms of storage space, account for almost 70% of the total size. It is notable that a large portion of the data set is contained in files without extension or with the .log extension; they represent more than 57% of the whole data set. Apart from this fact, the information was evenly distributed over the whole set of files.

We explored this large data set with an 8 KB expected chunk size which enabled us to reduce the potential number of chunks generated. All the percentages of similarity dropped. Although the percentage of similar data in fully identical files was 14%, the value obtained using the content-defined chunking strategy was only slightly over 20%. The advantages of using the content-defined chunking method are minimal if we consider that the fixed size blocking scheme offered a value of 17.52%.

According to our experiments, storage space used in unique chunks for this data set would be 88.26 GB and 91.59 GB, using the content-defined chunking method and fixed size blocks respectively. The compressed tar version of this data set claimed 49 GB of disk space.

We investigated separately the two main categories of files in our data set in terms of storage space used: .log files and files without extension. Using the content-defined chunking strategy .log files and files without extension showed values of around 0.3% and 30% of identical data respectively. Files without extension represent a considerably large amount of data that is difficult to characterise. However, they presented better levels of correlation compared to those obtained for the whole data set (20%). When the fixed size blocking strategy was used on files without extension, the percentage of identical data reached a value of 24.5%.

On the other hand, files with the .log extension offered extremely low levels of similarity and none of the .log files analysed were identical. It negatively influenced the sharing levels of the whole data set. Pathological cases such as these may be difficult to foresee and handle given the limited information that file names in these two categories provide about their contents; no relationship can be inferred *a priori* just by looking at the file names.

4.5 Software Distributions

In this experiment, we looked at five successive Linux kernel source distributions. Initially, we ran the content-defined chunking method over one of the kernels (2.5.34) and then added successive kernel versions one by one; we recorded the results at each step. Furthermore, we analysed the three possible states of the distributions (tar.gz, .tar, and raw files).

When the files were in the tar.gz format the percentage of data in shared chunks was 0% in all cases. Table 9 presents the values obtained in the other formats (tar and raw files) with two different expected chunk sizes. The amount of information shared was substantially greater in tar files when the expected chunk size was smaller. Once more, this result suggests that a smaller expected chunk size increases the likelihood of content overlap.

Furthermore, the last column of Table 9 shows the values obtained when we explored the whole file content of the original files. Comparing these values with those obtained in the 4 KB expected chunk size over raw files, which was the best scenario, in none of the cases the difference is greater than 2%. This lead us to the conclusion that, although the content-defined chunking method efficiently found identical portions of data, their main source was in wholly identical files. As would be expected, no similarity was found among the files in their tar.gz and tar formats when the whole file technique was used.

Version	8 KB		4 KB		File
	tar	raw	tar	raw	raw
2.5.34	1.49	2.39	2.26	3.18	1.5
+2.5.35	43.42	94.46	57.41	95.43	93.8
+2.5.36	44.17	96.33	58.24	96.94	95.12
+2.5.37	44.62	97.09	58.85	97.71	95.31
+2.5.38	44.99	98.33	59.25	98.70	96.86

Table 9: Sharing pattern percentages in a succession of five Linux kernel distributions. A 48-byte sliding window size was used in this experiment.

Figure 4 shows the discrete cumulative distribution function of chunk occurrences that was obtained considering the five kernels in their raw state and using an expected chunk size of 4 KB. These results indicate that the kernel distributions are very similar. When an ordinary file system holds different versions of Linux kernels in its primal state, it is storing the same information almost as many times as versions it holds. Storing a Linux kernel in its primal state adds around 145 MB of data but at least 95% of this information is already contained in chunks of precedent versions of the kernel. Therefore, a hypothetical storage utility would add only 7 MB of new data if reuses the chunks already stored. However, distributing patches of the kernels in their compressed format continues being the most efficient method of propagation. For example, the largest patch in our set of kernels accounts for only 977 KB. Even if we uncompress this patch file, its size is smaller than the size of non identical chunks that were found using the content-defined chunking method. Using the content-defined chunking method a hypothetical storage utility would transmit 7 MB in new chunks while the size of the uncompressed patch is only about 3.7 MB.

However, if these five kernels are decompressed and untarred, then processed with the content-defined chunking method in order to eliminate repetitions, and finally compressed again, the resulting size is only 38 MB. This value is considerably smaller than the original 171 MB used by five tar.gz kernel files, and only slightly over the 34 MB of an individual compressed kernel.

4.6 Associated Overheads

System designers considering employing any of these techniques to reduce data duplication must be aware of computational and storage overheads. Computational overheads are due to the calculation of SHA-1 digests and Rabin fingerprints. Additional CPU time and memory is spent in maintaining the data structures that keep track of the chunks generated (a hash table in our case) and their reference counters.

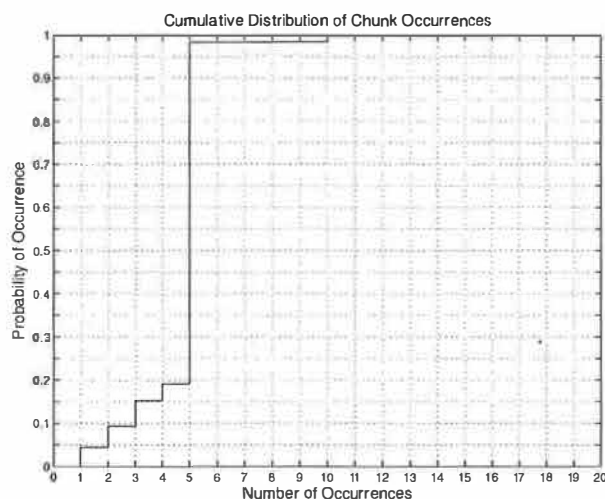


Figure 4: Discrete cumulative distribution function of chunk occurrences of five uncompressed and untarred kernels.

To compare the computational overhead in the three methods analysed, we created a 300 MB file containing random chunks of data taken from the data set corresponding to the Research Groups' files and then ran each of the methods on it. All the experiments were performed using a network-isolated machine with an Intel Pentium III 500 MHz processor. The content-based chunking method involves the generation of Rabin fingerprints and SHA-1 digests of the chunks. It took around 340 CPU seconds to process all the file. Around 76% of the total execution time was spent in tasks related to the computation of fingerprints over a sliding window. The fixed size blocking method took approximately 71 CPU seconds to compute all the necessary SHA-1 digests. Finally, the whole file approach used a total of 62 CPU seconds to calculate the digest. However, the reader should notice that our goal was to analyse data sharing patterns and potential storage space savings in diverse data sets; the prototypes were not implemented having performance as a compelling factor.

SHA-1 computations have been made extremely efficient. Commercially available hardware and operating systems' cryptographic services can be used to compute SHA-1 digests at very high speeds. Furthermore, Rabin fingerprints of a sliding window can be computed efficiently in software due to their algebraic properties, especially if the internal loop of the Rabin fingerprinting method is coded in assembly language [4]. An efficient implementation and computation of Rabin fingerprints on real-life data sets have already been reported [6].

Storage overhead is correlated to the number of unique blocks produced and the data structure used to keep track of their number of occurrences. As mentioned before,

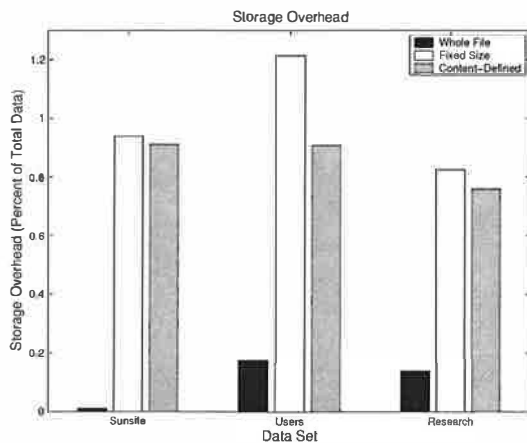


Figure 5: Storage overhead in three different data sets using a 4 KB expected chunk or fixed block size. Original data sets sizes: Sunsite=35 GB, Users=2.9 GB, and Research=21 GB

we used a hash table with this purpose. Figure 5 shows the storage space needed to store the hash table. The overhead has been computed for a 4 KB expected chunk size and fixed block size. It also compares these values with the overhead introduced by the whole-file strategy in which only one digest per file is required. In all the cases the amount of extra space required is small compared with the total size of the data set, but would pose a significant burden were it to be stored in memory.

5 Summary and Discussion

We found remarkable patterns in our results. The content-defined chunking algorithm was the best strategy to discover redundancy in the data sets studied. It consistently reported the largest amounts of data duplication. However, the fixed size blocking strategy also revealed useful levels of similarity. In the sunsite data set these values were considerably close to those obtained using the content-defined chunking method. For other data sets our results are similar to the numbers reported in [9] which revealed a more important difference in duplicated data detected by the content-defined chunking method. As may be expected, the whole-file approach was always at the bottom of the ranking.

The content-defined chunking strategy is specially efficient with potentially correlated data. We obtained high levels of similarity when the program was executed on data held by research groups (44.59%), and expanded source code distributions and software projects (98.7%). When this method was used on more diverse data sets, such as our sunsite.org.uk mirror and scratch directories, the similarity levels dropped (25% and 20% re-

spectively), and were noticeably closer to sharing levels found using a fixed size block approach (22.23% and 17.53% respectively). The whole file content approach reported modest levels of similarity with exceptionally high values in research groups' data (25%) and expanded source code distributions (96.86%). In data sets with a not-so-evident correlation such as the sunsite.org.uk mirror the similarity levels plummeted (5%).

In terms of storage space savings, the tar.gz version of the data sets consistently outperformed the other techniques. However, for systems that need to access and update separate files, probably in a distributed environment, compression is not easy to implement effectively. Chunk or block-based strategies such as the explored in this paper might be a better option in this domain.

In general, packed (i.e. rpm) and compressed data presented low levels of similarity; compression algorithms have already removed a degree of redundancy in the data set. Apparently, the storage space savings that can be obtained by decompressing a large number of arbitrarily selected files in order to remove data duplication from their expanded versions and finally compressing only non-identical chunks (see figure 3) does not justify the extra computational effort involved in this process.

The data structures needed to keep track of the extra information (SHA-1 block digests and reference counts) introduced a very small amount of storage overhead. The amount of extra storage required depends on the number of unique blocks managed by the hash table. As was expected, the whole file approach created only a very small number of unique entries in the hash table (i.e. one per file in the data set). The fixed size and content-defined methods produced comparable amounts of storage overhead, considerably greater than those exhibited by the whole file approach (see figure 5). A practical study of computational overheads incurred in each of the methods remains to be done. Our analysis, together with other experimental results [9], simply point out that there is an important amount of extra computation that has to be considered when using the content-based chunking method.

File access patterns should also be taken into consideration. Whole file content and fixed size blocking strategies present the disadvantage that file updates may lead to the recomputation of SHA-1 digests for large amounts of data. File updates under the whole file technique create the need to recompute the SHA-1 digest for the whole file. Using the fixed size blocking approach, any update that causes a shift at any position of the file will invalidate the SHA-1 digests for the rest of the blocks. As a consequence, reference counters of these blocks have to be decremented and SHA-1 digests have to be computed for the new blocks. However, a fixed size blocking scheme may offer the advantage that blocks can be page-

aligned and consequently improve memory performance as is pointed out by Sapuntzakis et al. [25]. On the contrary, updates in the content-defined chunking scheme are self-contained into the blocks where they occurred, thus SHA-1 digests are recomputed only for the modified blocks.

In light of our results, we consider there is no one method able to perform satisfactorily on all data sets. The extra processing and storage space required for each of the techniques, together with usage patterns and typical workloads of specific data sets can be decisive factors when deciding to employ these techniques. The fixed size blocking method offers high processing rates which make it a good candidate for interactive contexts. Despite the fact that the recomputation of SHA-1 digests could represent an inconvenience, especially under workloads in which file updates are common, our experimental results showed that the similarity patterns seen in passive data sets were relatively close to those obtained using the content-defined chunking strategy.

The question seems to be whether in practice the majority of the common blocks would remain valid after file updates and how often these updates occur. File access patterns [24, 30] indicate that file updates present significant locality: only a very small set of files is responsible for most of the block overwrites. Files tend to have a bimodal access, they are either read-mostly or write-mostly. Finally, an important percentage of files, even under different workloads, are accessed only to be read [24]. Considering this experimental evidence we feel persuaded to believe that, in the general case, an important number of blocks will remain valid for their lifetime.

Overall, the levels of data redundancy that can be identified using the fixed size blocking strategy are respectably high and sometimes close to those obtained using the content-defined chunking method. With file access patterns in consideration, the fixed size blocking strategy seems to be a sensible option for the general case; it is simple, acceptably effective, and quite efficient. We consider that the content-defined chunking method is justified only in contexts in which potential data repetition is high and the costs of not identifying redundant portions of data due to scattered updates throughout the file are high. Suppressing duplication at the file level still seems to be a good option especially where the amount of duplicated data is high and enclosed in a group of well connected machines [3]. System designers should take a decision based on the practical trade-offs between saving storage space, bandwidth consumption, and the computational and storage overheads necessary to support each of these methods; the results presented in this work can assist them in such a decision.

Acknowledgements

We thank Pablo Vidales, Tim Moreton, Tim Harris, the anonymous reviewers, and our shepherd Fred Douglass, for their suggestions and feedback. Calicrates Policróniades has a scholarship of the Mexican government through the National Council of Science and Technology (CONACyT).

References

- [1] S. Balasubramaniam and Benjamin C. Pierce. What is a File Synchronizer? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 98–108. ACM Press, 1998.
- [2] Kenneth Barr and Krste Asanovic. Energy Aware Lossless Data Compression. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, USA, May 2003.
- [3] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th Usenix Windows System Symposium*, August 2000.
- [4] Andrei Z. Broder. Some Applications of Rabin's Fingerprinting Method. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.
- [5] Andrei Z. Broder. On the Resemblance and Containment of Documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES'97)*, 1997.
- [6] Andrei Z. Broder. Identifying and Filtering Near-Duplicate Documents. In *Proceedings of Combinatorial Pattern Matching: 11th Annual Symposium*, Montreal, Canada, June 2000.
- [7] Calvin Chan and Hahua Lu. Fingerprinting Using Polynomial (Rabin's Method). Faculty of Science, University of Alberta, CMPUT690 Term Project, December 2001.
- [8] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making Backup Cheap and Easy. *ACM SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.

- [9] Timothy E. Denehy and Windsor W. Hsu. Duplicate management for reference data. Research Report RJ 10305 (A0310-017), IBM, October 2003.
- [10] Fred Douglass and Arun Iyengar. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of 2003 USENIX Technical Conference*, pages 113–126, San Antonio, Texas, USA, 2003.
- [11] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In *Proceedings of ER Workshop on Mobile Data Access*, pages 254–265, 1998.
- [12] Val Henson. An Analysis of Compare-by-Hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, USA, May 2003.
- [13] Richard M. Karp and Michael O. Rabin. Efficient Randomised Pattern Matching Algorithms. Technical Report TR-31-81, Center for Research in Computing Technology, Harvard University, 1981.
- [14] Purushottam Kulkarni, Fred Douglass, Jason LaVoie, and John M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of 2004 USENIX Technical Conference*, Boston, Massachusetts, USA, 2004.
- [15] Udi Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Francisco, CA, USA, 1994.
- [16] Tim Moreton. Pasta: a Distributed Scalable File System for the Pastry Routing Substrate. University of Cambridge, Computer Laboratory Part II Project, 2002.
<http://www.cl.cam.ac.uk/~tdm25>.
- [17] Tim D. Moreton, Ian A. Pratt, and Timothy L. Harris. Storage, Mutability and Naming in Pasta. In *Proceedings of the International Workshop on Peer-to-Peer Computing, Networking*, May 2002.
- [18] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-Bandwidth Network File System. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'01)*, pages 174–187, 2001.
- [19] Sean Quinlan and Sean Dorward. Venti: a New Approach to Archival Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, USA, 2002.
- [20] Michael O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [21] Michael O. Rabin. Discovering Repetitions in Strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 279–288. Springer-Verlag, Berlin, 1985.
- [22] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: Accountable Execution of Untrusted Programs. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 136–141, March 1999.
- [23] Sean C. Rhea, Kevin Liang, and Eric Brewer. Value-Based Web Caching. In *Proceedings of the 12th International Conference on World Wide Web*, pages 619–628. ACM Press, 2003.
- [24] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*, San Diego, California, USA, June 2000.
- [25] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [26] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, and Andrew Yu. Data Replication in Mariposa. In *Proceedings of the 12th International Conference on Data Engineering*, pages 485–494, 1996.
- [27] Neil T. Spring and David Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00)*, pages 87–95. ACM Press, 2000.
- [28] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.
- [29] Unison, File Synchronizer.
<http://www.cis.upenn.edu/~bcpierce>.

- [30] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'99)*, pages 93–109, 1999.
- [31] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transaction on Information Theory*, IT-23(3):337–343, May 1977.

Notes

¹<ftp://sunsite.org.uk>

²The maximum number of blocks obtained for any single data set in all of our experiments was $n \approx 18 \times 10^6$. We indexed these blocks using the first $b = 64$ bits of their SHA digests. The probability of having one or more collisions is given by $1 - (1 - 2^{-b})^n$. This small probability can be neglected in our experimental results.

Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying

Yu-Chung Cheng^{†*}, Urs Hölzle[‡], Neal Cardwell[‡], Stefan Savage[‡], and Geoffrey M. Voelker[‡]

[†]*University of California, San Diego*

{ycheng,savage,voelker}@cs.ucsd.edu

[‡]*Google*

{urs,ncardwell}@google.com

Abstract

The performance of popular Internet Web services is governed by a complex combination of server behavior, network characteristics and client workload – all interacting through the actions of the underlying transport control protocol (TCP). Consequently, even small changes to TCP or to the network infrastructure can have significant impact on end-to-end performance, yet at the same time it is challenging for service administrators to predict what that impact will be. In this paper we describe the implementation of a tool called *Monkey* that is designed to help address such questions. Monkey collects live TCP trace data near a server, distills key aspects of each connection (e.g., network delay, bottleneck bandwidth, server delays, etc.) and then is able to faithfully replay the client workload in a new setting. Using Monkey, one can easily evaluate the effects of different network implementations or protocol optimizations in a controlled fashion, without the limitations of synthetic workloads or the lack of reproducibility of live user traffic. Using realistic network traces from the Google search site, we show that Monkey is able to replay traces with a high degree of accuracy and can be used to predict the impact of changes to the TCP stack.

1 Introduction

There are many factors that conspire to limit the performance of Internet Web sites, including server response time, client workload, network characteristics and protocol behavior. However, each of these factors can vary considerably between sites and their interactions vary as well. Consequently, it is challenging to know *a priori* which of many potential optimizations will have an appreciable impact on a given site. It is rarely cost effective to test these alternatives exhaustively. Instead, administrators must make educated guesses based on their understanding of their site's unique demands.

In response to this problem, a variety of synthetic load testing tools have been developed [2, 3, 12, 19]. These

tools are largely based on analytic Web workload models that have been developed and validated against measurements in individual settings [4, 5, 9, 13, 14]. This synthetic approach has enormous benefits, since it is easy to set up and has the flexibility to explore a variety of workload parameters. At the same time, the underlying models require continual re-validation with up-to-date empirical data and, by their very nature, synthetic models are unlikely to match any *particular* site's workload with high accuracy. Moreover, few of these tools attempt to model the network conditions of the client population and therefore are poor predictors for changes in a site's implementation that are sensitive to network characteristics.

Another potential alternative is to implement protocol or network changes on a test server and then redirect a subset of real users to that server to evaluate the changes impact. This approach has the benefit of being highly realistic, but also suffers from the risk that some users will be negatively impacted. Ideally, this risk should only be taken when there is a strong reason to believe the change will offer a significant benefit.

Our work represents a middle road – offering a high degree of realism while not exposing real users to risks during testing. The tool we have developed, called *Monkey*, uses captured traces to accurately replay an emulated workload that is effectively identical to the site's normal operating conditions. To do this, Monkey infers delays caused by the client, the protocol, the server, and the network in each captured flow and then faithfully replays each flow according to these parameters to recreate the original workload. This approach allows a site administrator to recreate a real workload in a modified test environment – and thereby evaluate the impact of individual protocol, server or network optimizations. For example, a site with typically short data flows might wish to explore the effect of modifying TCP's initial congestion window setting, or investigate the benefits of using the TCP SACK or DSACK options to reduce spurious packet retransmissions.

To our knowledge, Monkey is the first tool of its kind and our initial experiences have been extremely positive.

*Cheng performed this work on an internship at Google.

Using traces gathered from the popular Google search site, we have been able to replay Google client workloads with high accuracy. Moreover, we have been able to successfully predict the impact of changes made to the Google TCP implementation on overall client response time.

In the remainder of this paper we discuss related work and then describe the design challenges and trade-offs in the Monkey system. We then describe the details of Monkey's trace collection and trace replay components and finally discuss our experiences using Monkey at Google to predict the impact of protocol changes.

2 Related Work

In his 1999 HotOS paper, Mogul offers a general indictment on the statement of systems benchmarking in use today. He argues strongly that relevant benchmarks must predict absolute performance in a production environment, rather than simply focusing on quantified, repeatable results in a carefully constructed laboratory setting [11]. Unfortunately, this goal has proved elusive in practice and few tools available today can offer strong predictions about future performance.

Most existing HTTP load testing tools, such as SPECweb99 [2], WebStone [19], Web Polygraph [3], httpperf [12] are based on synthetic models of Web traffic [4, 5, 9, 13, 14]. These models are developed analytically and then validated experimentally with measurement studies. In particular, such tools are focused on creating realistic traffic mixes as a function of overall load – a role for which they have been very successful. However, these tools typically are run on local area networks and ignore the impact of variable wide-area network characteristics or protocol interactions with different client operating systems.

An exception is the Nahum et al. study [14], which investigated the impact of WAN conditions on WWW server performance by combining synthetic experiments with an emulated wide-area network [14]. While their experiments only included static HTTP transactions, they still found that variations in round-trip time (RTT), packet loss rate, as well as different TCP versions (Reno, NewReno, SACK) had significant impact on end-to-end response time. However, Nahum's study did not attempt to replicate the workload or network conditions of a *particular* Web site and reflected the impact of a particular synthetic parameterization.

The open-source tcpreplay tool represents a very different approach to this problem [20]. Tcpreplay takes a packet dump and replays each recorded packet without transport or upper protocol knowledge – typically to exercise firewall and security systems. Although it can also be used to replay traces against a server, tcpreplay does not separate out network, client and server conditions and therefore it will not reflect the impact of any changes to the test environment.

A slightly more advanced approach is found in the

tcplib[7] tool, which simulates TCP applications (TELNET and FTP in particular) through a combination of deterministic application characteristics combined with statistical modeling of user behavior. The authors observed that wide-area TCP/IP traffic cannot be accurately modeled with simple analytical expressions, but instead requires a combination of detailed knowledge of the end-user applications and measured probability distributions of user workloads. Unlike Monkey, tcplib does not reproduce particular traces, but generates traffic according to the general characteristics revealed in a set of measurement experiments.

Zhang et al. [21] studied flow rates with traces captured from a large backbone ISP and discovered that short flow rates are over 90% application-limited or limited by slow-start behavior. Most Google flows fall into this category, where HTTP response time is highly correlated to server application delay and client slow-start behavior. Barford et al. [6] studied the cause of delays in general HTTP flows using a similar analysis, and under synthetic loads found that long response times are mainly contributed by server and client delays. These results are generally consistent with our traces of Google.¹ While we were not able to directly reuse the analyses from these studies, their approach informed our inference techniques.

3 Design

Monkey is designed to emulate a real workload by emulating client behavior (e.g., request timing, client delays, and protocol implementation), server behavior (e.g., service delay and protocol implementation), as well as network conditions (e.g., RTT, bandwidth and loss rate). Our current prototype is particularly focused on evaluating how changes in server implementation – particularly the TCP stack – affect HTTP response time.

Monkey consists of two distinct components: *Monkey See* and *Monkey Do*. *Monkey See* captures TCP packet traces at a standalone packet sniffer adjacent to the Web server being traced (in our case on a network tap in front of a Google search server). It then performs offline trace analysis to extract observable link delays, packet losses, bottleneck bandwidth, packet MTUs, and HTTP event timings. This connection information is stored in a database to be used in the subsequent replay step.

Monkey Do consists of a network, client, and server emulator residing on the same LAN. The network emulator is responsible for emulating the characteristics of client upstream and downstream links. The client emulator models client request timings and protocol behavior and directs its packets through the emulated virtual links. The server emulator models server delays and protocol behavior. Finally,

¹The most significant differences result from Barford's use of Linux clients, whereas Google client population is primarily Windows based. Windows clients delay ACKs during slow start leading to significant "artificial" delay for short flows

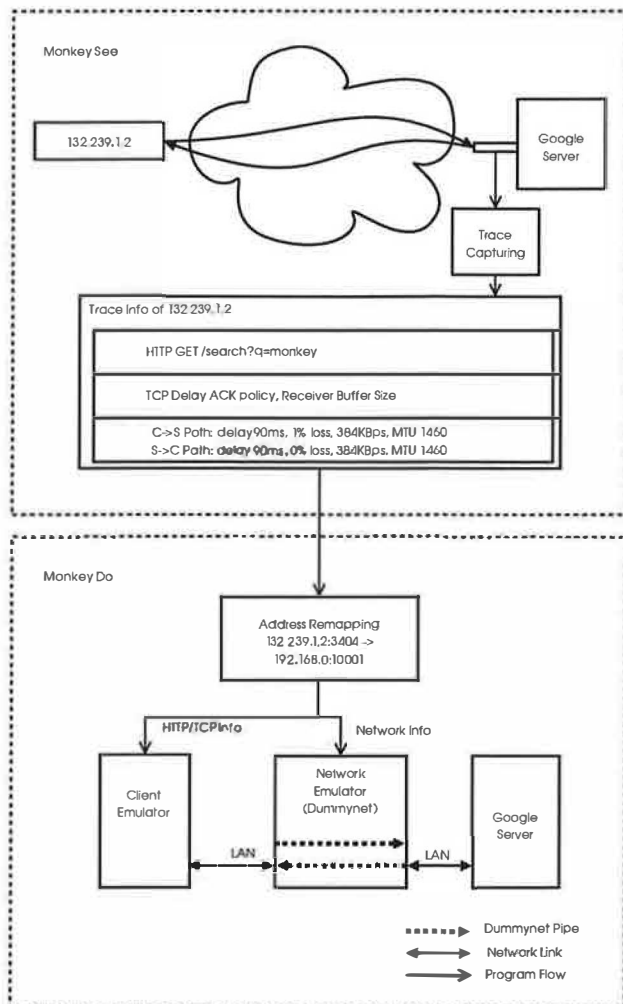


Figure 1: The Monkey Architecture. Monkey consists of two components: Monkey See and Monkey Do. Monkey See traces and analyzes TCP connections, while Monkey Do replays the connections by emulating both client and network behavior.

a packet sniffer at the test server captures traffic during a replay experiment so that it can be analyzed and compared to the original trace.

In the remainder of this section, we describe the Google service environment examined in our study and the design challenges and implementation details.

3.1 Assumptions

In its current form, Monkey is not a completely general tool. We have built Monkey in the context of the Google service environment and consequently we have been able to make simplifying assumptions based on our knowledge of this context. While most of these assumptions are common to any popular Web site, it is prudent to understand them before exploring the details of Monkey's design and implementation.

1. *High-performance local network.* Monkey collects traces directly in front of, not inside, the Google server cluster. The delay between the sniffing host and server end is less than 0.5 ms and Monkey ignores this delay in RTT and bandwidth estimations.
2. *Short flows.* Because most Google flows are rather short, ranging from 3–20 data packets, Monkey assumes client downstream and upstream network path dynamics, such as RTT and loss, do not change during the lifetime of a connection.
3. *No reverse-path congestion.* Using passive TCP analysis on server-side traces, it is not possible to perfectly infer exactly which packets are lost. Consequently, Monkey assumes that all losses occur on the server to client path. Similarly, Monkey assumes that queuing and congestion primarily occur on the server to client path. This is consistent with our bandwidth measurements in Section 3.2. Moreover, in a previous measurement of loss distributions to and from popular Web sites, Savage [17] found that over 90% of losses happen on the client downstream link and suggests that our assumption is reasonable.
4. *Well-provisioned servers.* We assume that the server cluster is well provisioned with processing capacity. In particular, we assume that the server will never queue packets for longer than a millisecond, and that delays longer than this are caused by application delays (e.g., search) or TCP flow/congestion control. This assumption holds true for the servers we used, but may be invalid in less well-provisioned infrastructures.
5. *Flow independence.* We assume that bottlenecks are independent and disjoint – individual connections do not interfere with each other. While in the real world it is possible that connections might share the same bottleneck link, our model would not capture such interactions during replay. Given the dominance of “last mile” bottlenecks, the small size of each flow, and the rarity of concurrent searches per user, we do not think this assumption is unreasonable. The largest potential exception is client proxies, but in our experience large proxies are also well provisioned with network bandwidth. However, if this assumption were violated, our replay might underestimate the typical response time.

3.2 Monkey See

Monkey See captured packet traces using `libpcap` [1] on a PC host equipped with two Gigabit Ethernet cards (one each for inbound and outbound traffic). Because we use the standard interrupt-driven Linux kernel to acquire packets, high data rates can overwhelm the host (in fact, during our trace collections at Google our network cards dropped

an average of 90,000 packets per second). To manage the packet drop rate, we reduce the amount of data captured by sampling flows with randomly selected values for the last octet. In our experiments we sub-sampled the traffic by a factor of 90 using this technique. Further, Monkey prunes connections that have incomplete connection handshakes, incomplete terminations (no FIN or RST) and incomplete data sequences. We also prune connections for which it is impossible to infer bandwidth estimates – typically extremely short flows that have no ACK pairs.

Once a trace is captured, Monkey See uses several analyses to extract key network and client characteristics and record these parameters into a replay database for each flow captured.

Path MTU, Delay and Loss. Client downstream and upstream path MTUs are extracted from the MSS TCP options contained in the client and server SYN packets respectively.

Monkey estimates the path propagation delays by halving the minimum RTT estimate. Since we are primarily concerned with overall response time potential propagation asymmetries are irrelevant. We use the minimum RTT because low bandwidth links, such as dial-up modems, can have very large variability in queuing delay. The minimum RTT is typically estimated between the server SYN-ACK and the first client ACK since packet queuing delays are usually small at this point.

As mentioned earlier, we assume the upstream path (ACK channel) has no loss. The downstream path (data channel) loss rate is estimated by counting the percentage of server data retransmissions. However, retransmissions may be spurious and cause Monkey to overestimate the loss rate. To address this problem, we employ the following heuristic. Assuming that packet reordering is rare, any duplicate acknowledgment sent in response to a retransmission indicates that the retransmission was spurious. Therefore, we can compute the packet loss rate as the number of retransmissions minus the number of duplicate acknowledgments, divided by the number of total server packets sent. In an environment using the TCP DSACK option [8], we could disambiguate spurious retransmissions even in the presence of packet reordering, but it is not in use on Google servers.

Link Bottleneck Bandwidth. Monkey uses the packet pair technique to measure bottleneck bandwidth. As mentioned in Section 3, Monkey assumes that packet queuing and congestion only happen at the end of the server to client path – the path from client to server is never congested. Hence, packets sent in a burst remain back-to-back when they arrive at the bottleneck link, e.g., modem, DSL lines, etc. If the client acknowledges packets immediately, Monkey obtains the bottleneck queue bandwidth by measuring the ACK time spacing. Sariou's probe tool [16] uses a similar technique, but Monkey measures all possible ACK

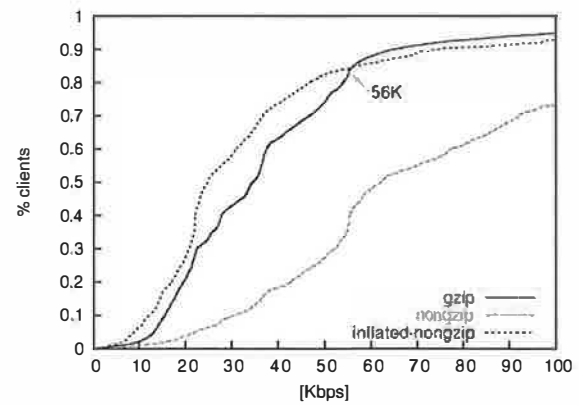


Figure 2: CDF of estimated bandwidth of dialup clients from level3.net. Over 80% are below 56 Kbps. The non-gzipped connections over-estimate application-level bandwidth, while the inflated-nongzip line reflects the adjustment of a compression factor of 2.5. The gzipped connections are not affected by modem compressions.

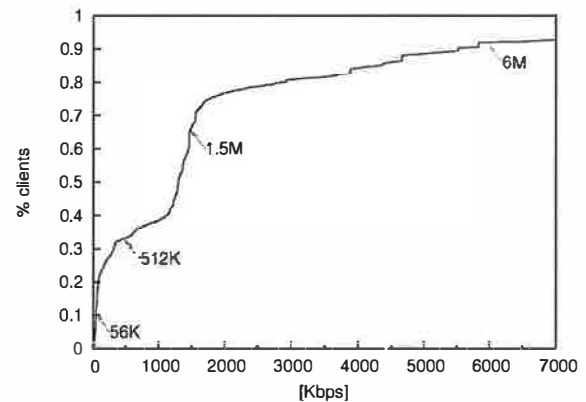


Figure 3: CDF of estimated bandwidth of DSL users from pac-bell.net. 9% have less than 56 Kbps and 90% connections have bandwidth beyond the maximum rate 6 Mbps. Note that time is measured with millisecond granularity, therefore bandwidth estimates over 1.5 Mbps (1460 bytes/1ms) can have large errors.

A challenge in using this technique is that TCP can delay an ACK for up to several hundred milliseconds [18] while waiting to receive another packet. This delay can lead to either under-estimates or over-estimates depending on the timing. Therefore Monkey discounts bandwidth estimates taken on ACK pairs that cover sequence numbers corresponding to data packets sent in separate bursts. On average, we are able to extract approximately three or four bandwidth estimates for each connection.

We have run two unit-tests to verify Monkey's bandwidth estimation. The first test measures the dial-up bandwidth for clients from level3.net with DNS hostnames containing the keyword dialup. We noticed that many of these connections present higher effective bandwidth than

cause for this discrepancy is that modern modems employ on-line compression. Since we cannot infer the actual compression ratio, we adjust the over-estimated bandwidth with a conservative compression factor of 2.5 for transactions containing non-gzipped HTML text [10]. As shown in Figure 2, after this adjustment, Monkey estimates that 80% of connections have bandwidths less than 56 Kbps. The second test measures the ADSL bandwidth for hostnames containing the keyword `dsl` from `pacbell.net`. As shown in Figure 3, 88% of connections have bandwidths between 56 Kbps and the highest subscriber speed of 6 Mbps. Higher rates in the graph represent over estimates.

TCP Delayed ACK Policies. When replaying a connection, it is critical to understand the behavior of the client-side TCP implementation. In particular, for short flows one of the most critical parameters is the delayed ACK policy. Since most Google flows (indeed, most Web flows in general) are short and never exit TCP's slow start phase, the rate of client ACKs may dominate the overall HTTP response time equation. In particular, while the Linux TCP stack does not delay acknowledgments when it believes the sender is in slow start, Windows clients delay ACKs uniformly. Monkey infers the delayed ACK policy in slow start by comparing the number of ACKs and data packets observed before the first loss occurs or the connection ends.

TCP Receiver Buffer Size. In addition to delayed acknowledgments, small TCP receiver buffers at the client can significantly limit server response time due to blocking on TCP flow control. Monkey records the advertised receiver window (RWIN) in the first HTTP request packet as the client buffer (we do not record the RWIN value contained in the initial SYN packet because Windows clients frequently use a small RWIN value in the SYN and then increase RWIN upon sending the HTTP request). Note, due to time constraints, the current implementation of Monkey Do does not emulate client TCP receiver window size. However, in an offline analysis we have determined that fewer than 0.4% of connections are limited by TCP flow control in our traces. For server applications with larger flows we expect that a better emulation of client buffering would be necessary.

TCP Connection Termination. Some client TCP implementations, notably Windows, send TCP RST to close connections rather than using an explicit FIN handshake. Consequently, it is ambiguous if a RST is due to an abort or a normal TCP close. Monkey assumes that if a RST is delivered after the end of an HTTP response, the client has received the data and intends to close the connection. Note that, since the choice of RST or FIN does not affect the HTTP response time, we have not emulated the RST termination behavior in Monkey Do.

HTTP messages. Monkey records the content and the timings of each HTTP message from the original trace. Notice that we do not need to decompress or parse the contents of the messages since Monkey only replays up to the HTTP

level.

3.3 Monkey Do

Monkey Do uses a client emulator, a network emulator, and a server emulator to replay traced connections, as shown at the bottom of Figure 1. All emulators run on separate machines on a well-provisioned LAN.

3.3.1 Network emulator

The network emulator uses Dummynet [15] to recreate the network conditions that were inferred from the trace during replay. For each HTTP connection, Monkey creates two Dummynet "pipes" as the forward and backward path with the corresponding delay, loss, and bandwidth inferred from the original trace. Currently Dummynet can support approximately 4000 simultaneous pipes (unique and independent network paths) while still forwarding packets between the client and server emulator with less than a millisecond of delay added.

Monkey configures the Dummynet pipe queue lengths based on the suggested value in [15]. For modem connections with bandwidths less than 56 Kbps, both the uplink and downlink pipe queue lengths are 5 packets. For DSL and cable modem connections have bandwidths ranging from 56 Kbps to 6 Mbps, Monkey sets the uplink queue to 10 packets and the downlink queue to 30 packets [15]. For the remaining high-speed connections, Monkey uses the default queue length of 40 packets.

At the start of replay, Monkey reads connection information from the replay database. For each connection, Monkey creates a new replay TCP connection identifier (source IP, source port, destination IP, and destination port) and maps it to the original TCP connection identifier. This mapping enables the client and network emulators to associate replayed connections initiated at the client emulator with the appropriate emulated network conditions at the network emulator. Figure 1 shows an example mapping for single connection.

Since Dummynet cannot do MTU emulation, Monkey uses the `TCP_MAXSEG` socket option in our replayed client and server sockets to model MTU in our replay network.²

3.3.2 Client emulator

The client emulator replays client HTTP requests in sequence by establishing connections to the server emulator through the network emulator. For each connection, Monkey creates user-level sockets using the same TCP addresses as the replayed TCP connections. It then configures the TCP receiver buffer size and delayed ACK

²Since Linux often uses the internal path MTU instead of the user-specified `TCP_MAXSEG` socket option, we patched the kernel to always obey the user-specified TCP MSS.

policies as recorded in Monkey Do (Section 3.2) using the TCP_RCVBUF and Linux-specific TCP_QUICKACK socket options, respectively.

To accurately emulate client behavior, Monkey needs to determine client event timings such as the connection start and termination times, and the HTTP request time. Since Monkey only uses server-side traces, it infers the timing of events at the client based upon an estimate of the one-way network delay (half the measured RTT). It models the client connection time as the SYN packet time in the trace minus this network delay, the HTTP request time as the first client packet minus network delay, etc.

3.3.3 Google server emulator

Finally, we describe our server emulator, how it emulates the HTTP behavior of a Google server interacting with a client, and how we parameterize it to model HTTP performance during replay. This is necessary because we cannot rely on a production server to accurately replicate the trace's application-level delays due to changes in the contents of Google's search result cache at different points in time.

The behavior and performance of a Google server fundamentally depends upon the nature of a Google search request and response. A Google search request is usually short (1–3 packets). A Google search response is usually longer (3–15 packets) and consists of two parts: a short, 3 KB search-independent Google header (the Google search form), and then the search results (see Figure 5a). Figures 5b and 5c illustrate packet timings of two typical Google search flows. In these figures, the x-axis shows the time at which packets are sent and received by the Google server and the y-axis shows the relative packets sequence number from the start of the connection. The small hash marks connected by lines show data packets for the HTTP response sent by the server, and the large crosses show the time and ACK sequence number of client ACKs received at the server. Figure 5b shows a typical uncompressed HTTP response consisting of 15 response packets and 8 client ACKs. Figure 5c shows a typical gzipped HTTP response consisting of only 4 response packets and 4 client ACKs; note that the use of compression significantly reduces the number of packets exchanged between the server and client.

Figure 4 shows the interaction of a Google client and server as a time line of packet exchanges. To reproduce the server behavior in the replay, we measure the server delays in the original trace and emulate them in the replay. First the client establishes the connection and sends the HTTP request. The server typically spends several milliseconds processing this request, a period of time we call the *response delay*. Next, the server simultaneously queries the appropriate database and sends out the Google header to the client. Therefore, we can measure the *response delay* as the difference between the timestamps of the last request

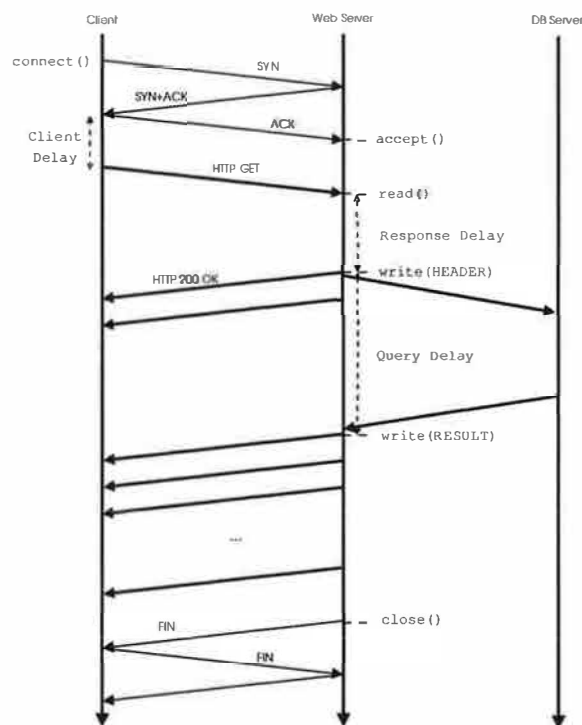


Figure 4: Time line of packet flows and major events in a Google search replay. First the client connects to the server. After the handshake, the client might delay a small interval before it sends out the HTTP request. Upon returning from `accept()`, the server initiates a `read()` to receive the HTTP request. When `read` returns, the server might not respond immediately if the original trace indicates a server response delay. Finally, the server writes the header, stalls to model the server search delay, sends out the search result, and closes the connection.

packet and the first response packet (which consists of the Google header).

Finally, the server waits until the result database returns the result – the *query delay*. Measuring the *query delay* can be challenging since the time at which the server starts to send search result data is not explicitly apparent from Monkey's vantage point. This information is not directly available from the packet payload, due to HTTP compression, nor can it be inferred purely from inter-packet delays since similar pauses can be caused by congestion or flow control. Instead, we analyze two aspects of TCP's behavior to differentiate application-level delays (attributed to query overhead) from network-level delays. First, since TCP packet delivery is ACK triggered³, if the client has acknowledged all outstanding server data packets, but the server has not sent more data, we can infer that the server has blocked waiting for application data. Second, since Linux's TCP implementation always packs data into packets of the Maximum Segment Size (MSS), we can tell that the server's sending buffer is empty after it sends a

³Google servers use a version of Linux that includes the NewReno variant of TCP's congestion control algorithm

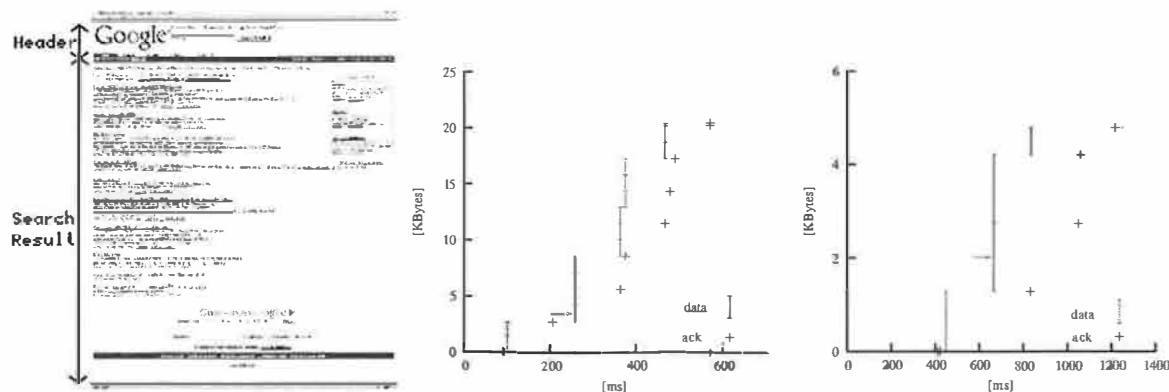


Figure 5: A Google Search and the corresponding TCP time-sequence graphs. (a) Google Search typically consists of a header and the search result. The header is sent immediately but the search result is generated after 20–500 ms. (b) The TCP time-sequence graph of a non-gzipped Google Search transaction. The first three packets consist of the header, followed by a search delay, then the search results. (c) The TCP time-sequence of a gzipped Google Search transaction. Most gzipped transactions uses 1 packet for the header, and 2 to 4 packets for the search result. In (b) and (c). The small hash marks connected by lines show data packets for the HTTP response sent by the server, and the large crosses show the time and ACK sequence number of client ACKs received at the server. The arrows point to the first data packet that contains search result.

sub-MSS-sized packet. Based on these two observations, we develop an algorithm to estimate the *query delay* as follows:

```

GOOGLE-QUERY-DELAY(tcp_segments)
1  s ← tcp_segments[1]
2  p ← s
3  c ← tcp_segments[2]
4  while c ≠ NIL
5      do if c.snd_time > p.snd_time and
6          c.snd_time > p.ack_time
7          then return (s.snd_time, c.snd_time)
8      else
9          if c.size < MSS
10         then return (s.snd_time, c.snd_time)
11     p ← c
12     c ← c.next
13

```

In the above algorithm, *tcp_segments*[*i*] refers to the *i*th segment sent by the server (i.e., *tcp_segments*[0] refers to the SYN/ACK packet). The key goal of the algorithm is to detect which is the last packet of the Google header and which is the first query result packet and return the difference in their timestamps. For each pair of sequential packets, starting with the first data packet, we check if there is significant delay between the packets – that they are not sent in the same burst – and that the previous packet was acknowledged before the current packet was sent. If so, we estimate query delay as the interval between the first and current data packets. For example, in Figure 5a, Monkey correctly estimates the *query delay* as the interval between the 1st packet and 3rd packet. However, if the query delay

is dominated by the round-trip time or if the client delays the ACK of the previous packet, then this test will fail. Instead we check if the current packet is less than full-sized as an indicator that this is the end of a header sequence (this heuristic will fail only if the size of last packet that contains the header is *exactly* MSS bytes). For example, in Figure 5b, the 1st packet is acked after the 2nd packet is sent, but 1st packet size is $1287 < 1460 = MSS$, hence the *query delay* is the interval between the 1st and 2nd packet. If neither test is satisfied then we consider the next pair of packets until completion.

After Monkey infers the server delay information from the trace, the server emulator uses it to mimic a Google server. It accepts requests from clients, emulates the server delays by idling, then writes HTTP responses. The server emulator runs on the same kernel as the Google servers so that kernel and protocol implementations are unchanged.

4 Methodology

In this section, we describe the types of connections in the Google search traces we use to evaluate Monkey, and the hardware platform we use for performing our experiments in Section 5.

4.1 Traces

For the experiments in Section 5, we use Monkey See to capture traces of TCP connections to the Google servers for replay with Monkey Do. Section 3.2 describes how Monkey See selects TCP connections to capture in a trace. Because these traces contain more than just Google search traffic, though, we also filter the connections based upon

application criteria as well. Since we are focused on Google search performance, we exclude all connections to all other Google services such as page ranks, images, news, or group search.

We also exclude search connections from ISPs that provide search services from Google through dedicated proxies. These ISPs have high network bandwidth and low network latencies. As a result, their connections are usually very short and the HTTP response time is dominated by the RTT.

Finally, of the remaining Google search connections, we exclude searches that use persistent connections (25% of search connections). Modeling persistent connections is a challenging problem since the HTTP response times are highly dependent on the TCP effective window size at the server at the end of the previous transaction, and remains future work.

4.2 Experimental platform

For the experiments in Section 5, the client emulator is one machine running Linux 2.4.20, the network emulator is another machine running FreeBSD 5.1, and the server emulator is a third machine running a Google Linux kernel. All machines have Pentium 4 2.66 GHz CPUs and 1 GB of main memory, and are connected by a dedicated 100Mbit Ethernet switch.

5 Experiments

In this section, we perform two experiments to evaluate the effectiveness of Monkey's replay features. First we evaluate the ability of Monkey to accurately replay traces from a Google server on the Monkey client, network, and server emulators, demonstrating that Monkey can accurately reproduce HTTP connection response times for a large fraction of traced connections. Then we demonstrate Monkey's ability to predict the performance of a server optimization.

5.1 Replay validation

We start by evaluating how well Monkey can reproduce the behavior and performance of Google search transactions. To do this, we compare the performance of search transactions to a Google server with the same search transactions modeled by Monkey. In this experiment, the Monkey server emulator uses the same kernel settings as the Google servers that originally performed the search requests. Assuming that Monkey models the client, network, and server behavior and performance accurately, then the HTTP response times of the replayed search transactions should match the response times observed in the trace. We define the HTTP response time as the interval from the time of the first byte of the request received to the time of the last byte of the response sent from the server.

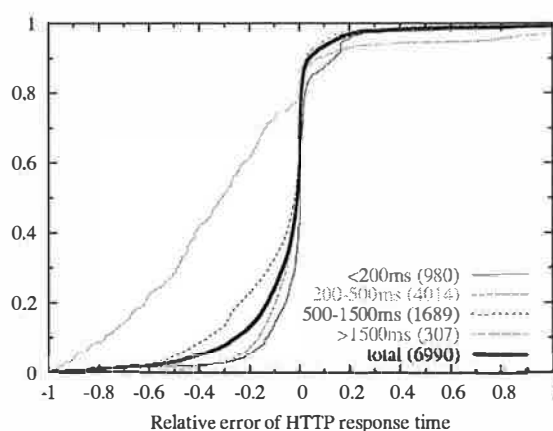


Figure 6: CDF of relative error in HTTP response time per connection for a trace of 6990 connections. Over 86% of connections have response time that are within $\pm 20\%$ relative error.

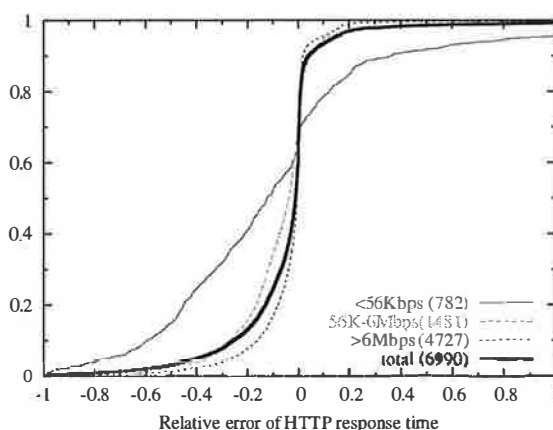


Figure 7: CDF of HTTP response time in replay and original trace, categorized in different bandwidth distributions.

For this experiment we use a trace of 6990 connections from 2:15–3:06pm on a weekday in November, 2003. To compare search transaction performance from the trace with search performance of the replay, we use a metric of relative error in HTTP response time. We compute relative error as the replay response time minus the trace response time, divided by the original response time. A relative error of 0.0 indicates the replay response time matched the trace response time exactly, a negative error indicates that the replay underestimated response time, and a positive error indicates that the replay overestimated response time.

Figure 6 shows the CDF of the relative error in HTTP response time across all replayed connections. The figure shows CDFs for all connections (“total”), as well as subsets of connections categorized by their HTTP response time; the number in parentheses in the legend label of these CDFs shows the number of connections in the category. For example, the dark solid CDF curve shows the relative error of the HTTP response time for all connections whose re-

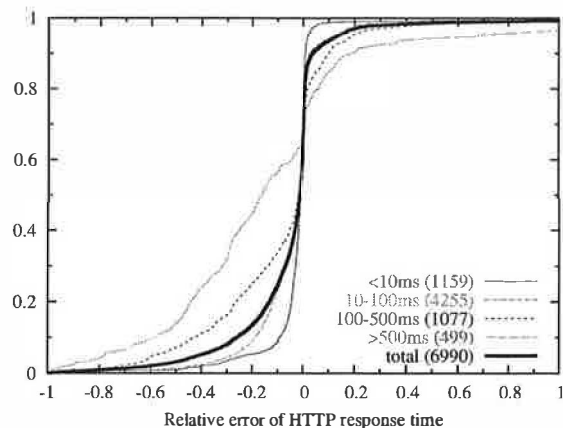


Figure 8: CDF of HTTP response time in replay and original trace, categorized in different minimum RTT distributions.

sponse times were less than 200 ms (“< 200ms”), and that there were 980 out of 6990 (14%) such connections.

From this graph we make a number of observations. First, Monkey performs reasonably well in reproducing HTTP response times when replaying the traces. Over 86% of the connections replayed within $\pm 20\%$ relative error. Second, when Monkey replay performance differs from the original trace, it tends to underestimate response times. Over 60% of replayed connections had a negative relative error. Third, Monkey’s ability to accurately replay connections correlates with the HTTP response time of the original connection. Monkey replays connections with fast response times more accurately than those with slow response times. For example, over 94% of connections with response times less than 200 ms had a relative error of $\pm 20\%$. And Monkey’s accuracy progressively degrades for slower connections. At the other extreme, only 29% of connections with response times greater than 1500 ms had the same relative error. We discuss this issue further below.

To study Monkey’s replay ability from different perspectives, we also correlate Monkey’s relative error to estimated bottleneck bandwidth and minimum RTT. Figures 7 and 8 show the CDFs of relative error in HTTP response time according to connections categorized by their estimated bottleneck bandwidths and minimum RTTs, respectively, of the original connections in the trace. From Figure 7, we see that Monkey’s replay accuracy also correlates with bottleneck bandwidth. Monkey does well for high-bandwidth connections, but progressively worse for low-bandwidth connections. Similarly, Figure 8 shows that Monkey does well with connections with low RTT and progressively worse for connections with higher RTT. Given the results shown in Figure 6, the results in these graphs are not too surprising since bottleneck bandwidth and minimum RTT also correlate strongly with HTTP response time: higher bandwidths and smaller RTTs result in smaller HTTP re-

sponse times.

In Figure 6 we saw that, when Monkey does not replay a connection accurately, it tends to underestimate connection response time. By manually inspecting various original and replayed TCP flows, we found that Monkey’s replay tends to have a more aggressive delayed ACK policy than the connections from the Windows clients in the trace, which together totaled over 90% of all connections. As a result, Monkey’s replayed connections will tend to perform faster than the original connections. Recall from Section 3.3.2 that Monkey uses Linux to emulate the clients in a trace. The Linux delayed ACK timeout on average is less than the Windows delayed ACK timeout. Linux provides a special TCP_QUICKACK setsockopt option to control ACK timeouts, but the kernel does not always obey the option setting and may still send an immediate ACK even when Monkey disables it. Further, Linux always sends out an ACK after it receives two consecutive in-sequence packets, but Windows may send out only one ACK for data bursts of four packets. As a result, the mean RTT in the replay is likely to be smaller than the mean RTT in the original for these connections.

5.2 Predictive replay

Next we evaluate Monkey’s ability to *predict* the performance of optimizations made to a Google server on a given client workload. In this experiment, we (1) trace the performance of a client workload on an original Google server, (2) trace the performance of an equivalent client workload on an optimized Google server, and then (3) use Monkey to replay the workload from the original Google server on our server emulator modified with the same optimization. By comparing the performance of the optimized Google server trace with the performance of the replayed unoptimized trace on the optimized server emulator, we can evaluate Monkey’s ability to predict performance of server modifications using trace replay.

For this experiment, the optimization that we make to the Google server and server emulator is to increase the TCP initial congestion window from 1 to 3. This optimization makes TCP more aggressive in sending data, thereby decreasing overall HTTP response time.

Ideally, we would like to use the same client workload on both the unoptimized and optimized Google servers in steps (1) and (2). However, it is impractical to do so. For example, we could not shadow the same workload simultaneously on two Google servers that differed in their initial congestion window but were otherwise exactly the same in terms in state and load. Instead, we use two equivalent workloads to the unoptimized and optimized Google servers. These workloads do not have the same TCP connections, but their overall distributions of connection performance (HTTP response time) are effectively identical. As a result, although we cannot compare workloads on

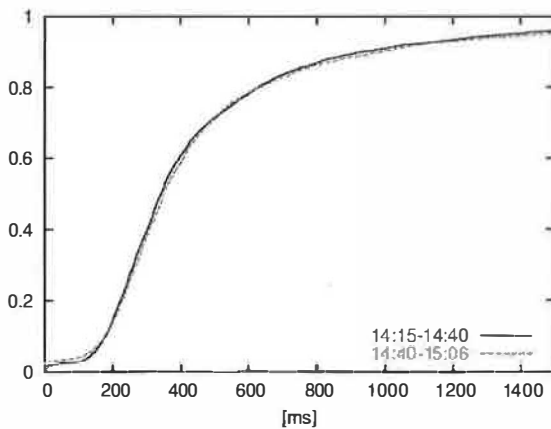


Figure 9: CDF of HTTP response times for two halves of a trace collected from 2:15–2:40pm and 2:40–3:06pm on a weekday in November, 2003.

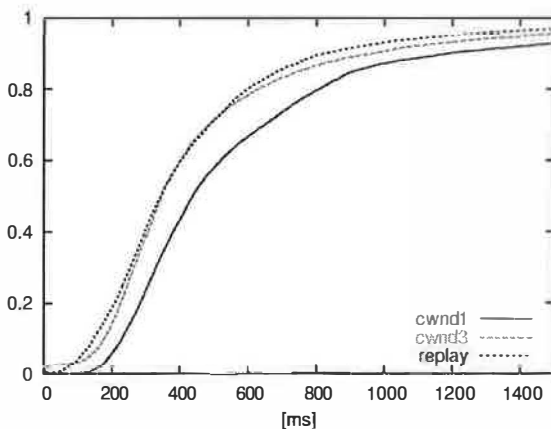


Figure 10: CDF of HTTP response times for traces t_{cwnd1} , t_{cwnd3} , and t_{replay} . Traces t_{cwnd3} , t_{cwnd1} , and t_{replay} have 7109, 6923, and 6841 connections, respectively.

a connection-by-connection basis, we can compare their overall distributions.

To validate this approach, we use a trace to a Google server from 2:15–3:06pm on a weekday in November, 2003. We divide the trace in half in time, from 2:15–2:40pm and 2:40–3:06pm, and compare the distributions of HTTP response time for both halves of the trace. Figure 9 shows the CDFs of the HTTP response time distributions of the trace halves. From the figure, we see that the distributions are nearly identical. For the purposes of this experiment, we therefore consider two relatively short workload traces taken immediately after each other in time to be equivalent workloads in terms of their HTTP response time distributions.

To evaluate Monkey’s ability to predict performance, we began with a trace t_{cwnd1} of a client workload to a Google server with its TCP initial congestion window set to 1. We recorded trace t_{cwnd1} from 1:30–2:15pm on a weekday in

November, 2003. We then changed the TCP initial congestion window of the Google server from 1 to 3, and immediately recorded a trace t_{cwnd3} of the client workload from 2:15–3:06pm (the trace used in Figure 9 above). We then used Monkey to replay the t_{cwnd1} trace on the server emulator running a Google kernel with a TCP initial window of 3, resulting in a trace t_{replay} of replayed connections.

We evaluate the ability of Monkey to predict the performance of an optimized server using a trace from an unoptimized server by comparing the distributions of HTTP response times in traces t_{replay} and t_{cwnd3} . The closer these distributions are, the better Monkey is at predicting the performance of this server optimization on an unoptimized client workload.

Figure 10 shows the CDFs of the HTTP response time distributions for traces t_{cwnd1} , t_{cwnd3} , and t_{replay} . Comparing the distributions of traces t_{cwnd1} and t_{cwnd3} , we see that increasing the TCP initial congestion window decreases HTTP response time, effectively shifting the distribution left over 100 ms. Recall that t_{replay} is a replay of t_{cwnd1} on a Google server emulator with TCP initial congestion window set to 3, the congestion window value of trace t_{cwnd3} . Comparing the distributions of t_{replay} and t_{cwnd3} , we see that the distributions match well within the 200–550 ms response time range, demonstrating that Monkey can predict performance well for this optimization in this range of client connections.

Recall from Section 5.1 that Monkey underestimates HTTP response time for connections that originally experienced relatively large response times in the trace. As a result, we expect the t_{replay} distribution to also slightly underestimate the t_{cwnd3} distribution at large response times. Furthermore, in this experiment, Monkey overestimates response times for connections with response times under 100 ms. This is because with an initial congestion window of 1, the connection is stalled waiting for a delayed ACK from the client. This delay overlaps, and hides, the actual search delay – leading to an overestimate.

6 Discussion

While Monkey is able to offer strong predictive power in the Google environment, an obvious question is how well this approach might generalize to other Web environments, or even further to other TCP applications. In general, there are several classes of problems presented by different environments: network dynamics, server emulation, and client analysis.

In its current form, Monkey makes several simplifications in its network model which, while appropriate for Google, may require significant extensions for other settings. For example, Monkey currently models packet losses as independent and identically distributed. In environments with single flows long enough to stress intermediate queues, the pattern of losses may be neither. Similarly, sev-

eral of the algorithms depend on regular acknowledgments returning from the client – assuming the reverse path congestion is rare. While many of these complexities can be addressed with better analysis algorithms, others represent inherent ambiguities. For example, the free-running nature of client-based delayed ACK timers makes the source of acknowledgment delay inherently ambiguous.

Another concern is the potential need for specific server emulators for each new application. However, this requirement is limited to those applications that have strong performance dependencies between sets of operations – such as the result caching employed in the Google system. In this setting the server emulator prevents these dependencies (e.g., all results from the trace being already present in the result cache) from skewing the results. However, for systems in which the performance distribution is “memoryless” and independent of the particular order and time of requests there is no need for a server emulation. Consequently, in most Web or content distribution environments, the trace can be directly replayed against the original server.

Finally, the current version of Monkey does not model client interactions. We cannot predict the impact of faster response times on future request arrivals. In general, it is unlikely that a complete end-to-end “closed-loop” analysis can be extracted purely from a TCP stream.

7 Conclusions

In this paper we have described a new tool called Monkey. Monkey collects live packet traces of TCP connections and then replays them to mimic the original session characteristics like network, server and client delays as well as packet loss and bottleneck bandwidth limitations. Monkey allows Web site administrators to quickly and easily evaluate the effect of different network implementations and optimizations in a controlled fashion, without the limitations of synthetic workloads or the lack of reproducibility of live user traffic.

Using realistic, large network traces from the popular Google search site we show that Monkey replays traces with a high degree of accuracy. We also demonstrate that Monkey can be used to predict the effect of changes to the TCP stack by showing that the measured impact of changes in the simulated environment closely corresponds to the impact of measurements taken on the actual system. In the end, we believe it is unrealistic to build a generic one-for-all TCP replay tool. But it is possible to build replay tool for specific applications as Monkey.

Monkey source is publicly available at: <http://ramp.ucsd.edu/monkey/>

8 Acknowledgments

We thank Vikram Asrani, Gerald Aigner, and the Google production team for their help in running experiments on Google servers and extracting traces from Google’s internal networks. We also thank the anonymous USENIX reviewers and our shepherd Srinu Seshan for their comments and insights.

References

- [1] Libpcap. <http://tcpdump.org>.
- [2] Specweb99 benchmark. <http://www.specbench.org/osg/web99/>.
- [3] Web polygraph. <http://web-polygraph.org>.
- [4] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.
- [5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS*, 1998.
- [6] P. Barford and M. Crovella. Critical path analysis of TCP transactions. In *Proceedings of ACM SIGCOMM*, January 2000.
- [7] P. Danzig and S. Jamin. tcplib: A library of TCP inter-network traffic characteristics. *Tech Report USC-CS-91-495, Computer Science Department, University of Southern California*, 1991.
- [8] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (sack) option for TCP. *RFC 2883*, July 2000.
- [9] S. Manley, M. I. Seltzer, and M. Courage. A self-scaling and self-configuring benchmark for web servers (extended abstract). In *Proceedings of ACM SIGMETRICS*, 1998.
- [10] P. Mitronov. Modem compression: V.44 against v.42bis. <http://www.digit-life.com/articles/compressv44vsv42bis/>.
- [11] J. C. Mogul. Brittle metrics in operating systems research. In *Proceedings of 7th Workshop on Hot Topics in Operating Systems*, January 1999.
- [12] D. Mosberger and T. Jin. httpf: A tool for measuring web server performance. In *Proceedings of First Workshop on Internet Server Performance (WISP)*, June 1998.

- [13] E. M. Nahum. Deconstructing specweb99. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution*, August 2002.
- [14] E. M. Nahum, M.-C. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on www server performance. In *Proceedings of ACM SIGMETRICS*, June 2001.
- [15] L. Rizzo. Dummynet. http://info.iet.unipi.it/~luigi/ip_dummynet.html.
- [16] S. Saroiu, P. K. Gummadi, and S. Gribble. Sprobe: A fast technique for measuring bandwidth in uncooperative environments. In *Proceedings of IEEE INFOCOM*, August 2001.
- [17] S. Savage. Sting: a TCP-based network measurement tool. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, October 1999.
- [18] R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison Wesley, 1994.
- [19] G. Trent and M. Sake. Webstone: The first generation in http server benchmarking, 1995. <http://www.sgi.com/Products/WebFORCE/WebStone/paper.html>.
- [20] A. Turner and M. Bing. tcpreplay. <http://tcpreplay.sourceforge.net>.
- [21] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *Proceedings of ACM SIGCOMM*, August 2002.

A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths

Ming Zhang, Junwen Lai
{mzhang, lai}@cs.princeton.edu

Arvind Krishnamurthy
arvind@cs.yale.edu

Larry Peterson, Randolph Wang
{llp, rywang}@cs.princeton.edu

Abstract

Recent work on Internet measurement and overlay networks has shown that redundant paths are common between pairs of hosts and that one can often achieve better end-to-end performance by adaptively choosing an alternate path [8, 28]. In this paper, we propose an end-to-end transport layer protocol, *mTCP*, which can aggregate the available bandwidth of those redundant paths in parallel. By striping one flow's packets across multiple paths, *mTCP* can not only obtain higher end-to-end throughput but also become more robust under path failures. When some paths fail, *mTCP* can continue sending packets on other living paths and the recovery process normally takes only a few seconds. Because *mTCP* could obtain an unfair share of bandwidth under shared congestion, we integrate a shared congestion detection mechanism into our system. It allows us to dynamically detect and suppress paths with shared congestion so as to alleviate the aggressiveness problem. *mTCP* can also passively monitor the performance of several paths in parallel and discover better paths than the path provided by the underlying routing infrastructure. We also propose a heuristic to find disjoint paths between pairs of nodes using traceroute. We have implemented our system on top of overlay networks and evaluated it in both Planet-Lab and Emulab.

1 Introduction

Recent work on Internet measurement and overlay networks has shown that redundant paths are common between pairs of hosts [28]. One can often achieve better end-to-end performance by adaptively choosing an alternate path other than the direct Internet path [8]. At the same time, stub networks are increasingly turning to multihoming to improve the reliability of their network connectivity [5]. The reliability is usually achieved by having sufficiently disjoint paths to the destinations of interest. Moreover, with the rapid growth of wireless coverage, mobile users can often have access to multiple communication channels simultaneously [17, 21]. All of the above means redundant paths are quite common between pairs of hosts.

Our goal is to design an end-to-end transport layer protocol (*mTCP*) which can not only aggregate the bandwidth on several paths concurrently but also enhance the robustness under path failures by taking advantage of those redundant paths. Compared with the conventional single-path

TCP flows, *mTCP* stripes a flow's packets across several paths. It can be viewed as a group of single-path subflows, with each subflow going through a separate path. We address a number of challenges in our attempt to develop such transport layer protocol based on TCP. First, the traditional congestion control mechanism in TCP needs to be modified to fully exploit the benefits that *mTCP* has to offer. A TCP flow does congestion control on the whole flow. If a flow happens to use a heavily congested path, severe packet losses on that path will keep the throughput of whole flow small so that it cannot make use of the available bandwidth on other better paths. *mTCP* does congestion control on each subflow independently so as to minimize the negative influence of subflows on poor paths.

Second, paths may fail during data transmission. *mTCP* should not stall as long as there exists one living path. It should be able to quickly detect failed paths and continue sending or retransmitting packets on living paths.

Third, when subflows of a *mTCP* flow share congested links, the whole *mTCP* flow can obtain an unfairly larger share of bandwidth than other single-path TCP flows, because each subflow behaves the same as a single-path TCP flow. To alleviate the aggressiveness problem, we integrate a shared congestion detection mechanism into our system so as to identify and suppress subflows that traverse the same set of congested links. Although in today's Internet, many congested or bottleneck links lie at the edge of the network which could limit the performance benefit of *mTCP*, this is likely to change with the growing popularity of high speed Internet access. In [6], Akella, Seshan and Shaikh measured a diverse set of paths traversing Tier-1, Tier-2, Tier-3 and Tier-4 ISPs. They found about 50% of the paths have bottleneck links located within ISPs or between neighboring ISPs. The available capacity of those bottleneck links are less than 50Mbps, well below 100Mbps Ethernet speed. Many of these paths have already limited the performance of well-connected nodes. Even when congestion does occur on edge links, using redundant paths can still improve the end-to-end robustness under path failures as described above.

Finally, there might exist many alternate paths between a pair of source and destination nodes. We want to select a small number of candidate paths for *mTCP* flows since it is impractical to use all the paths simultaneously. We use a heuristic to identify and select disjoint paths using tracer-

oute. This can minimize the possibility of shared congestion and concurrent path failures.

To the best of our knowledge, we are the first to implement and evaluate such a transport layer protocol, which can utilize redundant paths concurrently, in real systems. We try to provide a comprehensive design that addresses the inter-related issues of sub-flow congestion control, unfair use of congested links, path selection, and recovery from path failures. We believe that it is beneficial to tackle all of these issues in a single tightly-coupled system. For instance, suboptimal decisions from the path selection mechanism could be corrected by a mechanism that detects the use of shared congested links. Alternately, shared congestions could be detected easily by monitoring TCP events (such as fast retransmits) without requiring separate probe messages. Furthermore, the system could quickly recover from failures by maintaining and transmitting along multiple paths. Finally, a mTCP flow can passively monitor the performance of several paths in parallel and estimate their available bandwidths. The bandwidth estimates are typically more accurate than the estimates provided by the underlying overlay routing mechanisms. This in turn can help select better paths.

In this paper, we focus on improving the performance and robustness for large data transfers. Although most flows on the Internet are small, most of the traffic on the Internet is contributed by a small percentage of big flows [35, 13]. Therefore, improving the performance of such big flows is very important. Additionally, small flows can also benefit from mTCP because it can quickly detect and recover from path failures.

The rest of the paper is organized as follows: Section 2 will describe related work. Section 3 will discuss the specific design problems of mTCP in detail. Section 4 briefly describes the implementation of our system. Section 5 demonstrates the results from experiments conducted on PlanetLab [26] and Emulab [2]. Finally, Section 6 concludes.

2 Related Work

The general idea of using multiple paths in a network to obtain better performance has been explored in a number of different research efforts. We briefly discuss how our work relates to previous research in this area.

One area of related work is the use of striping [33] or *inverse-multiplexing* in link-layer protocols to enhance the throughput by aggregating the bandwidth of different links. Adishesu *et al* [4], Duncanson *et al* [12] and Snoeren [30] provide link-striping algorithms that address the issues of load-balancing over multiple paths and preserving in-order delivery of packets to the receiver. These efforts propose transparent use of link-level striping without requiring any changes to the upper layers of the protocol stack.

Another area of related work is the use of multiple

paths by transport protocols to enhance reliability [10, 24, 20, 9]. Banerjee [10] proposed the use of redundant paths in his *dispersity routing* scheme to improve reliable packet delivery for real-time applications. Nguyen and Zakhori [24] also propose the use of multiple paths to reduce packet losses for delay-sensitive applications. They employ UDP streams to route data whose redundancy is enhanced through forward error correction techniques.

The most directly relevant related work is the use of multiple paths for improving the throughput or robustness of end-to-end connections. Several application-layer approaches have been proposed to improve throughput by opening multiple TCP sockets concurrently [7, 14, 19, 29], but the multiple TCP connections utilize the same physical path. These approaches obtain an unfair share of the throughput of congested links and seem to primarily benefit from increased window sizes over long-latency connections. SCTP [32] is a reliable transport protocol which supports multiple streams across different paths. However, it does not provide strict ordering across all the streams, and it cannot utilize the aggregate bandwidth on multiple paths as we do. The systems that are closest to what is described in this paper is R-MTP [21] and pTCP [16]. R-MTP provides bandwidth aggregation by striping packets across multiple paths based on bandwidth estimation. It estimates the available bandwidth by periodically probing the paths. As a result, its performance greatly relies on the accuracy of the estimation and the probing rate. It could suffer from bandwidth fluctuation as shown in [16]. pTCP uses multiple paths to transmit TCP streams and describes mechanisms for striping packets across the different paths. They however assume the existence of a separate mechanism that identifies what paths to use for their pTCP connections, and they also do not address the issues of recovering from path failures or obtaining an unfair share of the throughput of congested links if the paths are not disjoint. Their study is also limited to simulations using ns[3].

3 Design

The design of our system seeks to satisfy three goals. First, given several paths, mTCP should be able to make full use of the available bandwidth on those paths. Second, when mTCP uses paths with shared congested links, it should be able to alleviate the aggressiveness problem by suppressing some of the paths. Third, when some paths fail, mTCP should quickly detect and recover from the failures.

3.1 Transport Layer Protocol

mTCP provides the same semantics to applications as TCP. It preserves properties such as reliability and congestion control. Because mTCP uses several paths in parallel, it has to decide how to stripe packets across the paths and how to manage congestion control for each subflow.

3.1.1 Congestion Control

In mTCP, all subflows share the same send/receive buffer. Packets are assigned sequence numbers in the same way as in TCP. But it does congestion control independently on each subflow. Each subflow maintains a congestion window as in TCP. The congestion window changes independently as the subflow adapts to the network state. When there are no packet losses in the subflow, it linearly increases. Upon detecting packet losses, it is halved. When timeout occurs, it is reset to one and the subflow enters slow-start.

mTCP strives to keep all subflows independent from each other. Suppose we had used only one global congestion window for the entire flow. The packet losses on any one of the paths will cause the global congestion window to be halved, thereby affecting the subflows on all paths. If one subflow happens to traverse a heavily congested path, it can keep the global congestion window small, and the other subflows will not be able to utilize the available bandwidth on other good paths. In certain situations, this can cause the throughput of the whole flow to be even lower than that of a single-path TCP flow on a single good path. This phenomenon was also studied in [16].

3.1.2 Estimating Outstanding Packets

TCP uses ($sndnxt - snduna$) to estimate the number of outstanding packets in the network. (For convenience, we assume packets are of the same size and use packets instead of bytes for discussion.) Here $sndnxt$ is the next packet to be sent and $snduna$ is the next packet for which an ACK is expected. It should be no more than the congestion window ($cwnd$). In mTCP, since packets are striped across different paths, we need to keep track of how many outstanding packets are in each path to ensure that the number does not exceed the $cwnd$ of that path.

Our mTCP is based on TCP SACK [22], which is an extension of TCP Reno. In Reno, the receiver only reports the greatest packet number that arrives in-order. But in mTCP, different paths have different latencies. Many packets can arrive at the receiver out-of-order. We want to accurately know which packets have been received, no matter they arrive in-order or out-of-order. Hence, we can compute the number of outstanding packets on each path, which is crucial to our congestion control. In SACK, sender maintains a scoreboard data structure to keep track of which packets have or have not been received. An acknowledgement (ACK) packet may carry several SACK blocks, where each SACK block reports a non-contiguous set of packets that has been received. The first SACK block reports the most recently received packet and additional SACK blocks repeat the most recently reported SACK blocks. The SACK blocks allows the sender to identify what packets have been newly received irrespective of whether or not the data packets arrive in-order.

We augment the scoreboard data structure so that it records the path over which each packet is transmitted or retransmitted. For each $path_i$, we maintain a $pipe_i$ to represent the number of outstanding packets on $path_i$. $pipe_i$ is incremented by 1 when the sender either sends or retransmits a packet over $path_i$. It is decremented when an incoming ACK indicates that a packet previously sent on $path_i$ has been received. New packets are allowed to be sent over $path_i$ only when $pipe_i < cwnd_i$. Retransmitted packets require special handling. Suppose the original packet is sent over $path_i$ and the retransmitted packet is sent over $path_j$. When the retransmitted packet is ACKed, the sender decrements both $pipe_i$ and $pipe_j$ by 1, because it represents two packets having left the network: the original one on $path_i$, which is assumed to be lost, and the retransmitted one on $path_j$, which has been received. We want to emphasize that the original and retransmitted packets do not have to be sent over the same path. We will discuss this in more detail in Section 3.1.4. Finally, if $path_i$ times-out, $pipe_i$ will be reset to 0.

3.1.3 Fast Retransmit

Since mTCP sends packets along several paths with different latencies, packets can arrive at the receiver out-of-order. This can cause duplicate acknowledgement packets (*dupack*), which will trigger fast retransmits. These fast retransmits are caused by packet reorderings and not by packet losses, therefore we want to avoid them. Although packets sent through different paths can be received out-of-order, packets within each subflow will still mostly arrive in-order. Each $path_i$ therefore maintains the following path-specific state: $dupack_i$, the number of *dupack* along that path and $snduna_i$, the next packet requiring an ACK. If an incoming ACK indicates the receipt of a packet sent through $path_i$ and if that packet is $snduna_i$, this packet is considered to be in-order within that subflow. If that packet is greater than $snduna_i$, $dupack_i$ is incremented by 1. When $dupack_i$ reaches $dupthresh = 3$, $path_i$ will enter fast retransmit and fast recovery.

3.1.4 Sending Packets

mTCP separates the decisions of when to send a packet, which packet to send, and which path to use to send the packet. The sender is allowed to send a new packet when there exists at least one $path_i$ satisfying $pipe_i < cwnd_i$. The packet to send is usually determined by $sndnxt$, which represents the next packet to send as in TCP. But if there is a $path_i$ with packets to retransmit, i.e. $path_i$ is in fast recovery, the sender has to retransmit those packets inferred to be lost before sending any new data packets. Once again the scoreboard is consulted to determine whether there are any such packets that need to be retransmitted. Otherwise, a new data packet referenced by $sndnxt$ will be sent.

Next, the sender needs to decide the path over which the packet will be sent. There may be several candidate paths. We associate a $score_i = pipe_i / cwnd_i$ with each $path_i$. We choose the path with the minimum score. This form of proportional scheduling results in a fair striping of packets and avoids sending a burst of packets on one path.

Because mTCP separates the decisions about when to send, which to send and which path to use for sending, it has more flexibility in striping packets. By postponing the decision about which path to use until just before sending out the packet, it can quickly adapt to dynamic variations in path characteristics. If a path encounters congestion or fails, its $cwnd$ will be reduced. The mTCP flow does not have to wait for the re-opening of the $cwnd$ on that path to retransmit the outstanding packets. It can retransmit those outstanding packets on other paths. We want to emphasize that, unlike the re-striping scheme used in pTCP [16], *our scheme will not retransmit packets that have already been received, because we can precisely infer missing packets from the scoreboard data structure.* In pTCP, such re-striping overhead becomes more significant when fast retransmit occurs more frequently.

3.1.5 Single Reverse Path

In our design, despite the fact that data packets are striped over several paths, all ACKs return over the same path. There are two reasons of using one path for ACKs. First, it is simple and it preserves the ACK ordering for all the subflows. If ACKs return from different paths, this may introduce ACK reorderings, which can further cause sender to misinterpret reorderings as packet losses and falsely enter fast retransmit on some path. Although using one reverse path could cause the forward and reverse path of each subflow to be asymmetric, it will not influence each subflow's normal operation. In fact, even for TCP flows between a source-destination pair, the forward and reverse path can be different because Internet routing is asymmetric. In [25], Paxson found 49% of the measured node pairs have asymmetric forward and reverse paths that visited at least one different city. We need to mention that the round trip time (RTT) of each subflow will be the latency of the corresponding forward path plus the latency of the single reverse path. Second, striping ACKs across different paths makes our system more complicated. Receiver has to maintain additional states about which ACKs going through which paths. We try to keep the receiver side as simple as possible, following the design principle of TCP. Besides that, using several reverse path will introduce ACK reorderings, which in turn will increase the burstiness of the sender.

The disadvantage of using one reverse path is the reverse path could be heavily congested or even fail. Although ACKs are small and normally do not cause congestion, we try to avoid congestion on the reverse path by selecting the best path among all the candidate paths with the help of underlying overlay router. This will be described in more

details in Section 3.3. We will discuss how to recover from path failures in Section 3.5.2.

3.1.6 Comparison with Multiple TCP sockets

We could have avoided implementing mTCP congestion control by opening separate TCP sockets for each path and then striping packets over different paths at the application layer [29]. We choose to modify TCP directly because it gives us more flexibility on striping data streams across multiple paths. mTCP can decide, for each packet, an appropriate path the packet should traverse and this decision is made just before the packet is sent out. This is especially useful for retransmitting packets on alternate path when the quality of paths changes dynamically or during path failures. Striping at the application layer across multiple sockets cannot adapt to changes in path quality. The pTCP study [16] has shown that such a scheme cannot fully utilize multiple paths when the number of paths exceeds two.

3.2 Shared Congestion Detection

When mTCP uses paths that are not completely disjoint and if some of the shared physical links are congested, the whole mTCP flow will obtain more bandwidth than other single-path TCP flows along those congested links, since each of the subflows behaves as a TCP flow. mTCP tries to alleviate the aggressiveness problem by detecting shared congestion among its subflows and suppressing some of them. Previous work [27, 15, 18, 34] on shared congestion detection is based on the observation that if two single-path flows share congestion, packets from two flows traversing a congested link at about the same time are likely to be either dropped or delayed. Rubenstein *et al.* [27] actively inject probing packets through the two paths to compute the correlation of packet losses or packet delays and thereby identify shared congestions.

Certainly, we can directly use one of the above approaches in our system since shared congestion detection is quite independent from other parts of the system. We however take a simpler approach based on the following observations. mTCP transmits a steady stream of packets through different paths. In this setting, there is no need to send probing packets. Instead, one can passively monitor the subflows by studying the behavior of the data packets. Furthermore, since individual packet drops will result in fast retransmits along the corresponding subflows, the sender can detect shared congestions by examining the correlations between the fast retransmit times of the subflows. Since data packets also double as probe packets and since there are a large number of data packets transmitted through a subflow, our passive monitoring strategy requires little overhead and generates a continuous stream of information resulting in fast detection of shared congestion.

3.2.1 Detecting Shared Congestion using Fast Retransmits

Let us focus on detecting shared congestion between a pair of subflows. For more than two subflows, we need to detect shared congestion between every pair of them. For abbreviation, if two subflows or paths share congestion, we say that they are *correlated*, otherwise, they are *independent*. We first assume that two paths have the same latency so that we do not have to worry about the time synchronization problem between them. Later, we will extend our algorithm so that it can deal with paths with different latencies.

Each time that a subflow enters fast retransmit, the sender records a timestamp in the subflow's list of fast retransmit events. After some time, we have two lists of timestamps, S and T , from two flows: (s_1, s_2, \dots, s_m) and (t_1, t_2, \dots, t_n) . Each timestamp represents a fast retransmit event. Then we try to match a timestamp s_i in S with t_j in T . If $|s_i - t_j| < interval$, we call (s_i, t_j) a match. Intuitively, a match means the two subflows enter fast retransmit around the same time. This also means packets from the two flows are dropped at about the same time, so it is likely they share the same congested link. We define $match(S, T)$ to be the maximum number of pairs (s_i, t_j) , such that s_i matches t_j . Please note that each s_i cannot be matched with multiple t_j . Finally, two subflows are considered to be correlated if:

$$ratio = \frac{Match(S, T)}{\min(m, n)} > \delta$$

$ratio$ is intended to identify what fraction of fast retransmits occur at about the same time in the two subflows. Since some of the fast retransmits are due to congestion on disjoint links, $ratio$ reflects the level of shared congestion. We consider two subflows to be correlated when $ratio$ is greater than some threshold δ .

Our method uses fast retransmits instead of individual packet losses to infer shared congestion. This is because when a data flow encounters congestion, there normally will be a burst of packet losses. All these losses are caused by one congestion period at some link. Therefore, the congestion period corresponds more directly to a fast retransmit other than any individual packet loss. We would like to declare (s_i, t_j) to be a match only when packets from two subflows are dropped at one link during the same congestion period. So $interval$ cannot be too small, otherwise even if s_i and t_j occur in the same congestion period, the system will not detect the match. On the other hand, $interval$ cannot be too large, otherwise the system would consider (s_i, t_j) to be a match even when they are not due to shared congestion. Although the shared congestion detection may not work well under active queue management schemes, most routers on today's Internet use drop-tail queues, which lead to periods of bursty losses during congestion. In [36], the authors find that 95% of the duration of bursty losses are less than 220ms. So $interval$

should be on that time scale. We will study how to choose $interval$ and δ in more detail in Section 5.4.

3.2.2 Estimating Convergence Time

We need to emphasize that our goal is to suppress correlated subflows in order to alleviate the aggressiveness problem. We need to detect shared congestion as quickly as possible. Other efforts focus more on the accuracy of shared congestion detection, and they may take several hundred seconds to reach a decision. This does not work well for our purpose, because a mTCP flow could have ended before shared congestion is detected.

Our algorithm works as follows. After some number of fast retransmit events have been observed, we will check for shared congestion between the two subflows. If there is shared congestion, we can suppress one of them. Otherwise, we will wait until the occurrence of the next fast retransmit to check for shared congestion again. The question we now address is determining the number of fast retransmit events that we need to observe before we start checking for shared congestion.

We use a heuristic to estimate the probability of two fast retransmit events from two independent flows accidentally occurring within a small period of time. Suppose the fast retransmit events of two subflows, S and T , are completely independent when two subflows are independent, we compute the average interval of two consecutive fast retransmit events in S : $interval_s = \frac{now}{m}$, where now is the current time when shared congestion detection is invoked. $interval_t$ is computed in a similar way. Then we define $p = \frac{2 \times interval}{\min(interval_s, interval_t)}$. Suppose $n \geq m$, we have $interval_t \leq interval_s$, and $interval = \frac{p}{2} \times interval_t$. For each s_i , if there exists a match t_j , s_i must be in the $(t_j - interval, t_j + interval)$. Because we assume s_i and t_j are independent events, the probability that s_i matches some t_j is roughly p . So the total expected number of matches is roughly $E(Match(S, T)) = pm$. Because $\min(m, n) = m$, we will misinterpret S and T to share congestion if $Match(S, T) > \delta m$. According to Chernoff bound [11]:

$$\zeta = Prob[(Match(S, T) > \delta m)] < e^{-mD(\delta||p)},$$

where $D(\delta||p) = \delta \ln \frac{\delta}{p} + (1 - \delta) \ln \frac{1 - \delta}{1 - p}$. So we need to wait for $m = -\frac{\ln \zeta}{D(\delta||p)}$ fast retransmit events to ensure that the probability of a false positive is less than ζ . We will see in Section 5.4 the convergence time is mostly within 15 seconds in our Emulab and PlanetLab experiments. We want to emphasize that even if false positive does occur, it will only degrade a mTCP flow into a single-path flow.

This heuristic might encounter problems when $\min(interval_s, interval_t) \leq 2 \times interval$. Because $interval$ is small (200ms in our experiments), this can only occur when a path is so heavily congested that fast retransmit happens almost every 400ms. The mTCP flows will try

to suppress such paths, because using them will not bring much benefit. This is discussed in Section 3.4.1.

Finally, when two paths have different latencies, there is a time-lag, L , between them. We estimate L by shifting one sequence, say T , by dt in time and calculating $Match_{dt}(S, T)$ on sequences (s_1, s_2, \dots, s_m) and $(t_1 + dt, t_2 + dt, \dots, t_n + dt)$ as described before. Because the L between two paths can be at most one RTT (RTT is the larger round trip time of the two paths), we go through all possible value dt in $(-RTT, RTT)$ incrementally using some fundamental step x , then choose dt that maximizes $Match_{dt}(S, T)$ as L . This is similar to calculate the correlation between two signals.

3.3 Path Selection

In the previous sections, we assumed that flows have a number of candidate paths. Now we describe how they obtain such information. We use Resilient Overlay Networks (RON) [8] as our underlying routing layer. RON is an application-layer overlay. When mTCP starts, it queries RON to obtain multiple paths between a source-destination pair. For each pair, RON provides the direct Internet path and alternate single-hop indirect paths through other RON nodes. With a RON of n nodes, there are totally $m = n - 1$ paths between each pair. RON uses a score to represent the quality of each path based on latency, loss rate or throughput. RON can effectively bypass performance failure or path faults by using an alternate path with higher score. In the following, we only use the throughput score.

Since m can be large (greater than 10 in our experiments), mTCP will only select at most k (5 in our experiments) paths from them. A single-path flow will normally select the path with the best score, which we call the *RON path*. mTCP could select the k best paths. But this simple strategy may select paths with many overlapping physical links. This leads to two disadvantages: First, paths are more likely to fail simultaneously, which is bad for the robustness. Second, paths are more likely to share congestion, which is bad for performance. To avoid these problems, we want to select sufficiently disjoint paths.

We use a heuristic based on traceroute to estimate the disjointness of paths. Using traceroute, we can obtain the IPs of the routers along a path and the latency of each physical link. Due to IP aliases, the same router might have different IPs in different paths. We use “Ally”, a tool from Rocketfuel [31], to resolve IP aliases and assign a unique IP to each router. Although some routers may not respond to traceroute probes and the alias resolution may not be completely accurate, we only use the traceroute information as a hint to estimate path disjointness and eliminate many of the significantly overlapping paths. We also rely on the techniques described in Section 3.2 to further detect shared congestion.

After alias resolution, suppose we have the IPs of two paths $X = (x_0, x_1, \dots, x_m)$ and $Y = (y_0, y_1, \dots, y_n)$. Let L

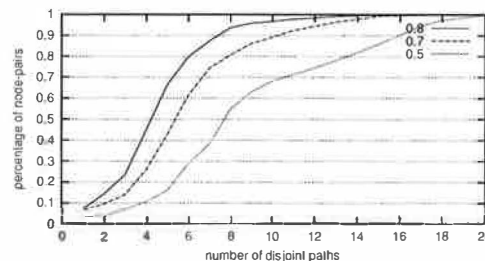


Figure 1: CDF of number of disjoint paths between node-pairs

be the set of overlapping links of X and Y , we define the overlapping between X and Y as: $Overlapping(X, Y) = \sum_{l \in L} latency(l)$. An alternative is to use the size of L to quantify the degree of overlap. We use latency instead because we hope to distinguish among different types of link. Most nodes on PlanetLab are connected through ethernet links to backbones. Those ethernet links usually have smaller latency than backbone links. Because the sharing of the local ethernet links are almost unavoidable, we focus on finding disjoint paths that traverse different backbone links. By using link latencies, $Overlapping(X, Y)$ will be mostly determined by the shared backbone links instead of ethernet links. This argument might not be true if nodes are connected through modem or wireless links that have high latency. Using traceroute to find disjoint paths is only suitable for small-scale overlay networks. As the number of nodes increases, we need a more scalable way to discover disjoint paths. In [23], Nakao, Peterson and Bavier propose to use BGP information to find disjoint Autonomous System (AS) paths, which incur little cost. Although disjoint AS paths are not as fine-grained as disjoint router-level paths, it would greatly simplify disjoint path search by providing a small set of promising candidate paths which we can further verify using traceroute.

Finally, we estimate the disjointness of X and Y by:

$$Disjoint(X, Y) = 1 - \frac{Overlapping(X, Y)}{Min(Latency(X), Latency(Y))}$$

We say that X and Y are disjoint if $Disjoint(X, Y) > \beta$. Using the disjointness metric between each pair of paths, we select at most k paths from m paths using a greedy algorithm as follows: (1) Initialize the set of selected paths to be empty. (2) Pick the path with the highest score from the set of m paths and check if it is disjoint from all the previously selected paths. (3) If so, select this path, otherwise pick the path with the next highest score and repeat step (2) until we find k paths or we have tried all m paths. The first selected forward path and the reverse path will always be the *RON path*, which is optimized for throughput in RON.

Figure 1 plots the cumulative distribution function (CDF) of the number of disjoint paths between 630 node pairs based on traceroute among 36 PlanetLab nodes that are used in our experiments. When β decreases, the number of disjoint paths between node-pairs increases. We want

a β such that there are sufficient number of disjoint paths which we can choose from while eliminating most significantly overlapping paths. When $\beta = 0.5$ (the value used in our experiments), 90% of node-pairs have more than 4 disjoint paths but less than 16 disjoint paths. If we use a larger β , many node pairs will not have enough candidate disjoint paths.

3.4 Path Management

3.4.1 Path Suppression

In mTCP, a subflow f_i on $path_i$ may be suppressed because of one of the three reasons: First, f_i shares congestion with another subflow f_j and its throughput $T(f_i)$ is lower than $T(f_j)$. This is because we want mTCP not to be too aggressive to other single-path TCP flows. Second, suppose f_j has the highest throughput among all the subflows and $T(f_i) < \frac{T(f_j)}{\omega}$. This is because $path_i$ is too poor and using it does not bring much benefit. Third, $path_i$ fails.

We define a family of mTCP flows, called MP_d flows. An MP_d flow will try to use at least d ($d \geq 1$) paths, which means we will not suppress any path because of shared congestion when the number of paths being used is less than or equal to d . For example, this avoids all paths getting suppressed in MP_1 flows. The value of d is a tradeoff between performance/robustness and friendliness. With a larger d , mTCP can obtain more bandwidth because it uses more paths. And it is more reliable because the probability that d paths fail simultaneously normally gets smaller as d increases. But it can be more aggressive to single-path flows under shared congestion. The aggressiveness problem can be alleviated by suppressing some subflows. But when there are only d subflows, no subflow would be suppressed. The actual value of d should be decided by the application. Applications that want higher performance and more reliability should choose a larger d . Applications that care more about friendliness should choose a smaller d . In our experiments, we choose $d = 1$ to demonstrate how much performance improvement mTCP can obtain without being too aggressive to other TCP flows.

3.4.2 Path Addition

An MP_d flow can dynamically add new paths because of two reasons: First, some paths that are not being used become better than those paths being used. Second, it is using less than d paths because some paths are too bad or have failed. mTCP will periodically update the information about all the paths by querying RON. If an unused path has much higher a score than a path being used, it can start using the new path. Then it runs the path suppression algorithm on all the paths to suppress any possible paths with shared congestion. By doing this, mTCP can gradually replace bad paths with good ones. This is especially useful for long-lived flows.

3.5 Path Failure Detection and Recovery

3.5.1 Failure Detection

mTCP may encounter path failures during transmission. If all the paths fail simultaneously, we call it a *fatal path failure*, otherwise we call it a *partial path failure*. We will focus on partial failures in this section. To recover from fatal failures, mTCP rely on the routing layer to establish new paths just like single-path flows.

When a path fails, the data packets sent over it will no longer be acknowledged (ACKed) because the packets have been dropped. We maintain one failure detection timer, $timer_i$, for each $path_i$. When a data packet sent over $path_i$ is ACKed, $timer_i$ is reset. $path_i$ is considered to have failed when $timer_i$ expires.

We need to decide a timeout value I_i for $timer_i$. On one hand, we want a small I_i so that failures can be detected quickly. On the other hand, I_i cannot be too small, otherwise it may misinterpret good path to have failed. The retransmission timeout (RTO_i) provides a good base for computing I_i . First, during timeout, the sender will go into idling and no packets will be ACKed in that period. So I_i should be at least greater than RTO_i . Second, several consecutive timeouts means either the path has failed or it is heavily congested. In either case, we would like to abandon $path_i$. So we choose $I_i = \chi RTO_i$. Here χ reflects how many consecutive retransmission timeouts mTCP is willing to tolerate before it consider a path to have failed. In our experiments, we choose $\chi = 2$, because we have observed that consecutive retransmission timeouts rarely occur on good paths. We should emphasize that even if a good path is misinterpreted as a failed one, it will only degrade the performance of mTCP to that of a single-path flow in the worst case. The path addition technique described in Section 3.4.2 allows us to reclaim a path if it had been previously misinterpreted to be a failed path.

3.5.2 Failure Recovery

We now describe how to recover from failure after $timer_i$ expires. Since all ACKs return over the same path, we call that path a *primary* path. The other paths are *auxiliary* paths. We need to distinguish between primary and auxiliary path failures. When an auxiliary $path_i$ fails, the sender will mark $path_i$ as failed and retransmit the outstanding packets of $path_i$ over other paths. When a primary path fails, the situation is more complicated. Because all the ACKs are lost, it may appear to the sender that all paths have failed. To deal with this problem, sender records the time π_i when $path_i$ is detected to have failed. Suppose at time *now*, the primary $path_p$ is also detected to have failed and let the timeout of $timer_p$ be I_p . We know that $path_p$ must have failed at some point between *now* - I_p and *now*. For an auxiliary $path_i$, if *now* - $I_p \leq \pi_i$, its failure is possibly due to the failure of $path_p$. In this case, we will change the status of $path_i$ to be active and the status of

$path_p$ to be failed. After doing this for all the paths, the sender starts to send packets over all active paths. During this period, these data packets serve as “probing” packets that solicit ACKs from the receiver. All timers are stopped to prevent any auxiliary path from being misinterpreted as failed due to the lack of an active primary path during this period. The receiver will also detect the primary path failure because it no longer receives any data packets over that path. Then it elects a new primary path and sends ACKs along that path in response to those “probing” packets from sender. It chooses the best path (based on the path score in RON) among all the active paths to be the new primary path. Later, when the sender receives the ACKs and knows that a new primary path has been elected, it restarts all the timers and proceeds as normal.

Typically, RTO_i is one second, therefore the I_i is two seconds. The total detection and recovery time will be between two and three seconds in most cases. The interruption due to partial path failures will be fairly short. Furthermore, partial path failure does not cause mTCP to stall since packets will continue to be transmitted through living paths. Since mTCP uses several paths concurrently and since it typically employs disjoint paths, the probability of fatal path failures is much lower than that of single-path failure. mTCP is therefore more robust than single-path flows.

4 Implementation

Our system is implemented at the user-level and is composed of a Portable User-Level TCP/IP stack (PULTI) and an overlay router/forwarder modified from RON. RON is an application-layer overlay on top of the Internet. PULTI and RON run in two separate processes and we change RON so that it can communicate with PULTI using UDP sockets and export the multiple paths between a source-destination pair. The whole system does not require any root privilege, which can easily be deployed on shared distributed platforms such as PlanetLab. Currently, it runs on Linux, NetBSD and FreeBSD.

PULTI is a full user-level TCP/IP stack based on FreeBSD 4.6.2. We extract the network-related code from the kernel source and wrap it with some basic kernel environment support, such as timing, timer, synchronization and memory allocation. We do not modify any network-related code. Because FreeBSD 4.6 does not support SACK, we also add SACK-related code in PULTI which is required by our system. OS dependent information is hidden by device drivers. With different device drivers, PULTI can send or receive through UDP socket, IP_QUEUE in Linux or divert socket in FreeBSD. PULTI provides standard socket interface and supports multiple applications through multithreading. It can query RON to learn about multiple paths between a source-destination pair. The mTCP code only affects a few files in PULTI. It can be

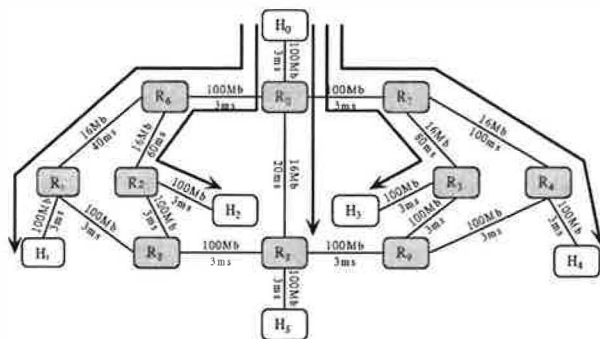


Figure 2: Topology of multiple independent paths on Emulab

easily moved into FreeBSD kernel.

5 Evaluation

5.1 Methodology

In this section, we validate our protocol in both emulation and real-world deployment. The emulations are run on Emulab [2], which is a time- and space-shared network emulator. Emulab consists of several hundred PCs, which can be configured to emulate different network scenarios. Users can specify parameters such as packet loss rate, latency, and bandwidth. While an experiment is running, the experiment gets exclusive use of the assigned machines. While Emulab provides a controlled environment for our experiments, we further conduct experiments on PlanetLab, a wide-area distributed testbed for running large-scale network services [26]. The experiments on the PlanetLab allow us to study our protocol for Internet settings, where latency, bandwidth and background traffic are more realistic and unpredictable.

5.2 Utilizing Multiple Independent Paths

In this experiment, we study whether mTCP can obtain the total available bandwidth over multiple independent paths. We use the topology in Figure 2 on Emulab. Because each PC in Emulab has four Ethernet cards, each node can have at most four links. There are six endhosts (H_i) and ten routers (R_j). RON is running on the six endhosts to construct an overlay network. All routers have drop-tail queues. The source and destination nodes are H_0 and H_5 respectively. Each of the remaining endhosts provides an alternate path. For example, we can use H_1 to construct an alternate path ($H_0, R_0, R_6, R_1, H_1, R_1, R_8, R_5, H_5$). So the topology contains five independent paths, which include one direct path and four alternate paths. We use the direct path as the reverse path for ACKs. The capacity of all the paths is 16Mbps and their RTTs vary from 52-147ms. The figure annotates each link with its corresponding bandwidth and latency. The arrows represent background flows. We use Iperf [1] to generate 25 TCP and 25 1Mbps UDP

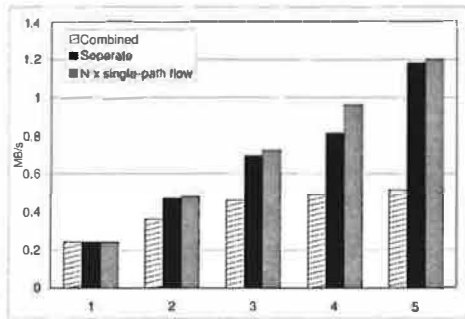


Figure 3: Throughput of mTCP flows with combined or separate congestion control as number of paths increases from 1 to 5

Path	Intermediate node	RTT(ms)
0	direct path	80.165
1	planetlab1.nbgisp.com	112.503
2	planet2.berkeley.intel-research.net	71.639
3	planet2.pittsburgh.intel-research.net	96.641
4	planet2.seattle.intel-research.net	90.305

Table 1: Independent paths between Princeton and Berkeley nodes on PlanetLab.

flows as background traffic, with 5 TCP and 5 UDP flows on each path. Each experiment runs for 40 seconds and the results are obtained by averaging three runs.

Figure 3 shows the results when the number of paths used by mTCP increases from 1 to 5. In this figure, “combined” represents mTCP flows with congestion control performed on the entire flow, “separate” represents regular mTCP flows with congestion control performed separately on each subflow, and “NxSingle-path flow” is the throughput of a single-path flow on one path multiplied by the number of paths. Because each path has the same available bandwidth, “NxSingle-path” throughput represents the ideal throughput of a mTCP flow. The results verify that mTCP can effectively aggregate the available bandwidth on multiple independent paths. The results also show that higher throughput can be achieved only when congestion control is performed for each subflow separately.

We conduct similar experiments on PlanetLab. We use one node in Princeton and one node in Berkeley as source and destination nodes. As shown in Table 1, the four Intel nodes serve as intermediate nodes for the alternate paths. We only use the four alternate paths in this experiment, because they do not share any congestion links. To verify this, we examined the traceroute data to find that any pair of the alternate paths only share the initial and final hops, which are unavoidable. The capacity of these two links are 100Mbps, which is far greater than the total throughput of the single-path TCP flows on these four paths. Therefore,

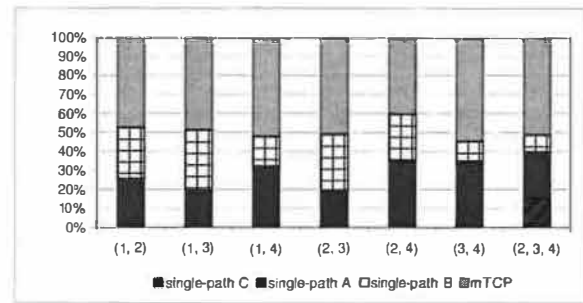


Figure 4: Throughput percentage of individual flows

Path	Intermediate node	RTT(ms)
0	direct path	80.165
1	planetlab02.cs.washington.edu	102.890

Table 2: Paths used in the failure recovery experiment.

we conclude that the initial and final hops are not congested and the four alternate paths are independent.

Each experiment measures the throughput of flows lasting for 60 seconds. The average throughput of three runs is reported. For convenience, we use $T(i)$ to denote the throughput of a single-path flow on $path_i$. Similarly, $T(i, j)$ denotes the throughput of a mTCP flow using $path_i$ and $path_j$. In Figure 4, (i,j) on the x-axis means $path_i$ and $path_j$ are used in that experiment. We first run single-path flows on $path_i$ and $path_j$ respectively, then run a mTCP flow on both paths simultaneously. The corresponding column compares the percentage that the throughput of an individual flow, $T(i)$, $T(j)$ or $T(i, j)$, contributes to the total throughput of these flows. Ideally, we expect $T(i, j) = T(i) + T(j)$, so the percentage of $T(i, j)$ should be around 50%. With the exception of the experiment involving $path_2$ and $path_4$, which suffered from unexpected bandwidth variations, the rest of the experiments indeed provide the expected throughputs. The last column in Figure 4 shows the result of the experiment using $path_2$, $path_3$, and $path_4$. Again, the net throughput of $T(2, 3, 4)$ is close to the sum of $T(2)$, $T(3)$ and $T(4)$. We have conducted experiments between different source-destination pairs on PlanetLab. The results are similar. We omit them due to space constraints.

5.3 Recovering from Partial Path Failures

Now we will study whether mTCP can quickly recover from partial path failures using experiments on PlanetLab. Because path failures on the Internet are unpredictable, we intentionally introduce failures by killing the appropriate RON agent. The source and destination nodes are still the Princeton and Berkeley nodes. The paths are shown in Table 2.

The two graphs in Figure 5 show how the congestion window ($cwnd$) of the primary and auxiliary paths changes over time. As shown in the first graph in Figure 5, the pri-

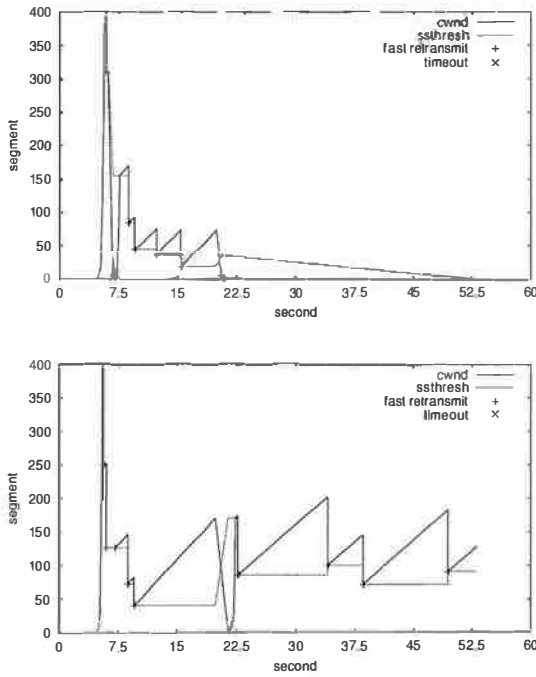


Figure 5: *cwnd* of primary/auxiliary paths, primary fails

primary path fails at about 20s. It is quickly detected so that the *cwnd* of the subflow on this path is reduced to 0. At the same time, the *cwnd* of the subflow on the auxiliary path also decreases to 0, because the auxiliary path was misinterpreted to have failed (as explained in Section 3.5.2). But a few seconds later, the subflow on the auxiliary path recovers from this false decision by restoring its *cwnd* to the previous value with slow start. Finally the auxiliary path becomes the new primary path and the whole flow proceeds using only one path. The behavior of mTCP during auxiliary path failures is similar, and we omit the corresponding results.

The total recovery time of mTCP during partial path failures is only about 3s, which is negligible for most applications. In contrast, a TCP flow will completely stall when its path fails, and it typically takes about 18s for RON to establish a new path. RON is optimized for quickly recovering from path failures. On wide area network that uses BGP to detect failures, recovery could take several minutes. Hence, mTCP is more responsive and robust than single-path flows.

5.4 Detecting Shared Congestion

In this section, we will evaluate shared congestion detection. We first use experiments on Emulab to study the behavior of our algorithm with different parameters in a controlled environment. Then we further validate it using experiments on PlanetLab. The topologies for the Emulab experiments are shown in Figures 6 and 7. Between the source node H_0 and the destination node H_2 , there is one

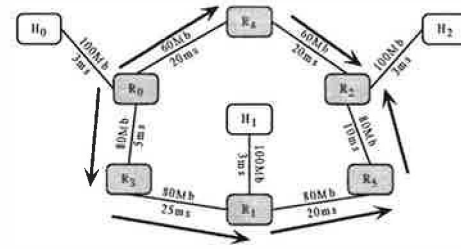


Figure 6: Two independent paths used in shared congestion detection

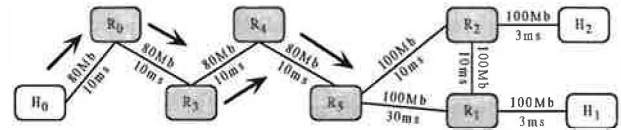


Figure 7: Two paths that completely share congestion

direct path and one alternate path through the intermediate node H_1 .

In Figure 6, The two paths only share the initial and final hops with link capacities of 100Mbps. We generate 12 TCP flows and 18 1Mbps UDP flows as background traffic, with 2 TCP flows and 3 UDP flows on each link between each pair of neighboring routers. With this scheme, we ensure that congestion only occurs on the links between pairs of routers and not on the links between endhosts and routers. As a result, the two paths (H_0, H_2) and (H_0, H_1, H_2) are independent.

In Figure 7, The two paths share the four links between H_0 and R_5 . We generate 8 TCP flows and 8 1Mbps UDP flows as background traffic, with 2 TCP and 2 UDP flows on each of the four shared links. By doing this, we ensure that congestion only occurs on the four shared links. As a result, paths (H_0, H_2) and (H_0, H_1, H_2) share congested links.

We run mTCP flows for 300s using the two paths in Figure 7. The results in Figure 8 compare the estimated *ratio* of shared congestion with different *interval* values of 5ms, 10ms, 25ms, 50ms, 100ms, 200ms, and 400ms. Each data point represents the average of five runs. As *interval* in-

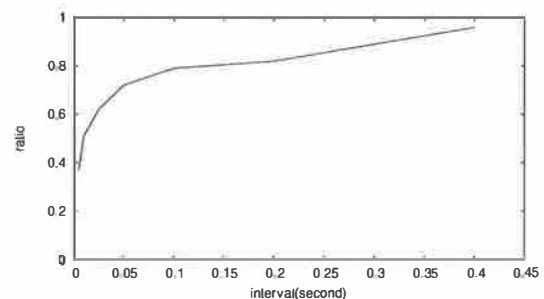


Figure 8: On two paths with shared congestion, *ratio* increases as *interval* increases

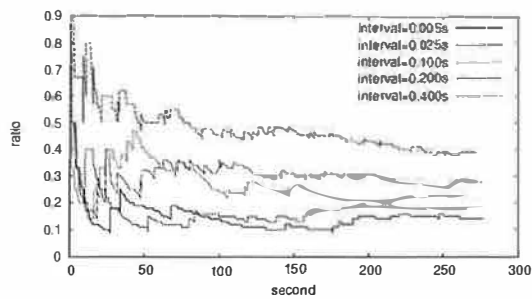


Figure 9: On two independent paths, *ratio* decreases faster when *interval* is smaller

creases from 5 to 100ms, *ratio* increases quickly from 0.4 to 0.8 as expected. When *interval* increases beyond 100ms, *ratio* only increases slightly. When *interval* is 400ms, *ratio* reaches 0.96. The ideal *ratio* is 1 because the two paths share all the congestion.

We next run mTCP flows for 300s on the two paths in Figure 6. The results are shown in Figure 9, which plots *ratio* over time for different *interval* values of 5ms, 25ms, 100ms, 200ms and 400ms. As explained in Section 3.2, a smaller *interval* will lead to smaller estimated values of *ratio*. At the end of the experiments, *ratio* drops quickly from 0.39 to 0.28 when *interval* decreases from 400 to 200ms. When *interval* decreases further, *ratio* drops more slowly until it reaches 0.14 when *interval* = 5ms. The ideal *ratio* is 0 because the 2 paths are independent. We also notice that the *ratio* curve for a smaller *interval* value decreases faster than that for a larger *interval*.

According to the above experiment results, an *interval* value between 100 and 200ms seems to balance the goal of minimizing both false negatives and false positives. Consequently, the *ratio* threshold δ should fall between 0.3 and 0.8. If it is less than 0.3, it is very likely to cause false positives when the *interval* is 200ms. If it is greater than 0.8, it can easily cause false negatives when the *interval* is 100ms. By setting *interval* = 200ms and $\delta = 0.5$, we successfully detect shared congestion between the two paths in all five runs for the topology in Figure 7. For the topology in Figure 6, no shared congestion is detected and the two paths are determined to be independent as expected.

Next, we go on to evaluate the shared congestion detection on PlanetLab. As explained in Section 3.2, by setting *interval* to be no less than the congestion period during which bursty losses occur, we can avoid false negatives. In [36], the authors find that 95% of the duration of bursty losses on the Internet are very short-lived (less than 220ms). By choosing an *interval* around that value, we should be able to avoid most false negatives. At the same time, the average time between consecutive fast retransmits is mostly on the order of several seconds or more, much greater than 220ms. (Otherwise, mTCP will suppress such path because the path is too lossy.) Therefore, this *interval* value will also allow us to avoid most false posi-

Path	Run 1	Run 2	Run 3
1 2	No	No	No
1 3	No	No	No
1 4	No	No	No
2 3	No	No	No
2 4	N/A	Yes	No
3 4	No	No	No

Table 3: Shared congestion detection for independent paths.

Path	Intermediate node	RTT(ms)
0	direct path	80.165
1	planetlab2.cs.duke.edu	96.138
2	planetlab2.cs.cornell.edu	100.382
3	vn2.cs.wustl.edu	92.267

Table 4: Paths with shared congestion on PlanetLab.

tives, as long as we wait for enough number of fast retransmits. In the following experiments, we report the results using *interval* = 200ms and $\delta = 0.5$.

We first need to choose paths such that we can be reasonably sure as to whether they share congestion or not. Then, we can compare the measured results with the expected results. We conduct two sets of experiments. The mTCP flow is running on a pair of paths for 60 seconds¹ in each experiment. Each experiment is repeated three times. We use the Princeton and Berkeley nodes as source and destination in all experiments, but we choose different pairs of paths in different sets of experiments.

In the first set of experiments, we use the four alternate paths in Table 1, where we know that all these paths are independent. The results are in Table 3. The first column shows the pairs of paths used by the mTCP flows. The remaining three columns show the results. A *No* means two paths are independent, a *Yes* means they share congestion, and *N/A* means one of the subflows is suppressed because its throughput is much lower than the other subflow before the end of the experiment. All the results in Table 1 conform to our expectation except the one false positive for using *path*₂ and *path*₄. As explained before, a false positive will only degrade the performance of the mTCP flow to that of a single-path flow.

The second set of experiments use the paths in Table 4. From traceroute, we know the underlying physical links of any pair of these paths are mostly overlapping, so they should share congested links. The results are shown in Table 5. The first column gives the pairs of paths used in the experiments. The following three columns give the time in seconds when shared congestion is detected in each run. The last column gives the average detection time. Shared

¹As explained in Section 3.2, the probability of false positive decreases very fast as the number of fast retransmit increases. We find that a 60 second period is long enough for our algorithm to converge.

Path	Run 1	Run 2	Run 3	Average
0 1	7.000	9.975	4.266	7.080
0 2	4.276	3.223	6.011	4.503
0 3	6.847	3.263	14.214	8.108
1 2	12.184	8.906	16.804	12.631
1 3	4.478	10.101	13.131	9.237
2 3	12.380	9.873	17.845	13.366

Table 5: Shared congestion detection for correlated flows.

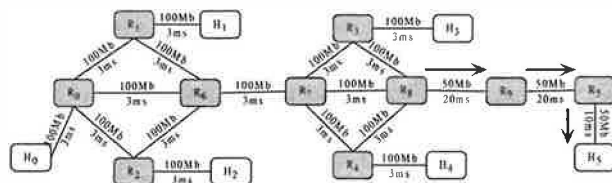


Figure 10: All paths share congestion in this topology

congestion is correctly detected in all cases.

Unlike other shared congestion detection algorithms, our algorithm seeks to minimize the detection time while maintaining a low false positive rate. In the second set of experiments, shared congestion is correctly detected mostly within 15 seconds. At the same time, such early decisions do not cause too many false positives in the first set of experiments.

5.5 Alleviating Aggressiveness with Path Suppression

In this section, we demonstrate mTCP can be more friendly to other single-path flows by suppressing its subflows that share congestion. We construct the topology of Figure 10 on Emulab. The source and destination nodes are H_0 and H_5 . There are one direct path and four alternate paths provided by the remaining four endhosts. Their RTTs are from 124ms to 133ms and they share the three links between R_8 and H_5 . We generate 12 1Mbps UDP flows as background traffic, with 4 UDP flows on each of the three shared links. By doing this, we ensure that all five paths share congestion. Each experiment runs for 300 seconds and the results

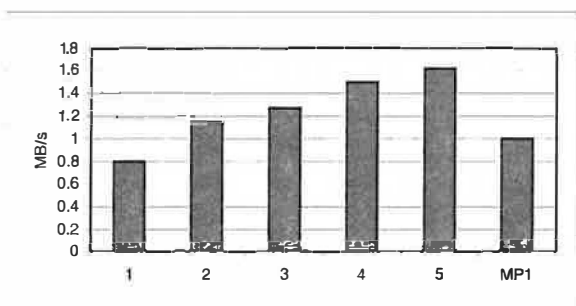


Figure 11: MP_1 flows are less aggressive than other mTCP flows

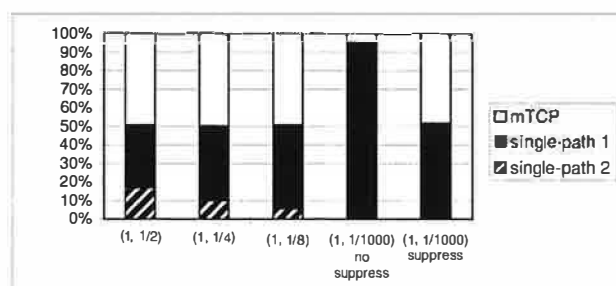


Figure 12: Path suppression helps avoid using bad paths.

are obtained by averaging three runs.

In Figure 11, the first five columns give the throughput of the flows when the number of paths being used increases from one to five. The first column is the throughput of single-path TCP flows. Under shared congestion, the mTCP flows become more aggressive as they use more paths. The sixth column shows the throughput of the mTCP flow with path suppression. Although it uses five paths in the beginning, it quickly detects shared congestion and suppresses all but one path. So its throughput is very close to that of a single-path flow and less aggressive than the flow using all five paths without suppression.

5.6 Suppressing Bad Paths

In this experiment, we demonstrate that mTCP can effectively aggregate the bandwidth of multiple paths with sufficiently differing characteristic, and path suppression can help avoid the penalty from using bad paths. We use the same topology as in Figure 2. The bandwidth of direct path is still 16Mbps. But the bandwidth of four alternate paths is 1/2, 1/4, 1/8 and 1/1000 of the bandwidth of the direct path. In Figure 12, $(1, 1/n)$ on the x-axis means the direct path and alternate path with $1/n$ bandwidth are used in that experiment. We first run a single-path flow on each path respectively, then run a mTCP flow on both paths. The corresponding column compares the percentage that the throughput of an individual flow contributes to the total throughput of these flows. Ideally, the throughput percentage of mTCP flows should be 50%. Figure 12 shows mTCP can efficiently utilize the aggregate bandwidth of two paths even when one path has only 1/8 the bandwidth of the other path. The mTCP flows in the last 2 columns use the direct path and the alternate path with 1/1000 bandwidth. Such a scenario could occur when a path becomes heavily congested or even temporarily fails. Using such bad paths can bring no benefit but impair the performance of the whole flow. Because most packets are lost along that path, it persistently causes timeouts. While packets can still be sent over the other path for some time, the flow will finally stall when the send/receive buffer is exhausted. As explained in Section 3.4.1, mTCP will suppress the paths with too low a throughput to avoid such penalty. (We choose $\omega = 10$ in our

Host Name	Host Name
planetlab2.millennium.berkeley.edu	planetlab2.postel.org
planetlab02.cs.washington.edu	planetlab2.lcs.mit.edu
planetlab-2.cs.princeton.edu	planetlab2.cs.ucla.edu
planetlab2.cs.uchicago.edu	planet.cc.gt.atl.ga.us
planetlab2.cs.duke.edu	pl2.cs.utk.edu

Table 6: The 10 endhosts used in the experiments that compare mTCP with single-path flows.

experiments.) This is confirmed by the last two columns which represent the throughput of mTCP flows with and without suppression.

5.7 Comparing with Single-Path Flows

We are going to compare three types of flows: single-path flows using direct Internet path (INET), single-path flows using *RON path* optimized for bandwidth (RON) and *MP₁* flows. *MP₁* flows will use multiple paths when there is no shared congestion. We use *MP₁* flows to demonstrate how much performance improvement mTCP can obtain without being too aggressive to other TCP flows. Table 6 shows the 10 nodes that serve as endhosts in an overlay network for this experiment. (We actually use a total of 24 nodes to form the overlay network, with the remaining 14 nodes only serving the role of packet forwarders.) For each source-destination pair, we transfer data for 40 seconds using each of the three types of flows. Each experiment is repeated three times and we report the average throughput.

The available bandwidth of the paths between the pairs of endhosts can be very high, because nine of them are connected to Internet2. We bypass those pairs with very high available bandwidth on the corresponding direct paths because: First, these paths are between pairs of nodes that exhibit shared congestion/bottleneck at the initial and/or final hops. Second, the bandwidth-delay products of the paths between such pairs of nodes are very large. The maximum send/receive buffer size of our user-level TCP implementation is 1MB and is not large enough to utilize the bandwidth on other alternate paths besides the direct path. We estimate the available bandwidth of a direct path between a source-destination pair by running a TCP flow for 10 seconds. If the measured throughput is less than 12 Mbps, we will use that pair for our experiments. Among the 90 pairs, we got 15 pairs that satisfy the above condition. We want to emphasize that we are not trying to study the popularity of independent paths with distinct points of congestion between node-pairs on the Internet; such topic has been studied by others [6]. Instead, we focus on demonstrating that mTCP can achieve better performance by taking advantage of such redundant paths.

Among the 15 pairs, *MP₁* flows achieve significantly higher throughput in 6 pairs, as shown in Figure 13. They achieve 33% to more than a factor of 60 better performance

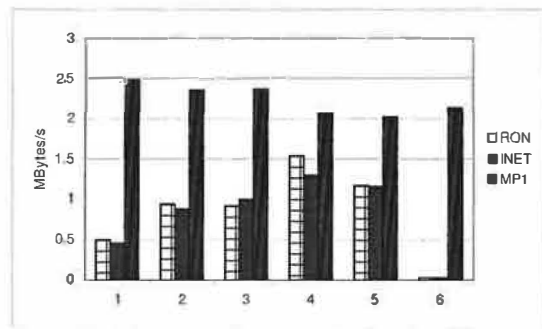


Figure 13: mTCP flows achieve better throughput than single-path flows

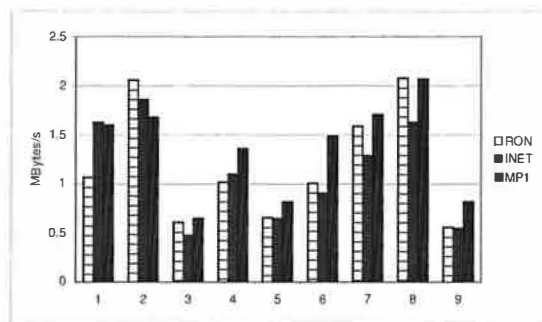


Figure 14: Throughput of mTCP and single-path flows is comparable

than single-path flows. We have to mention that *MP₁* flows only try to aggregate the available bandwidth on multiple paths when there is no shared congestion. Other *MP_d* ($d \geq 2$) flows would obtain better performance, but they are potentially more aggressive.

The performance improvement of mTCP does not solely come from bandwidth aggregation on multiple paths, it is also because mTCP can help select better paths than those provided by the routing layer, such as the direct path or the *RON path* optimized for throughput. RON estimates the available bandwidth of a path using $score = \frac{\sqrt{1.5}}{rtt\sqrt{p}}$. Here p is the packet loss rate and rtt is the round trip time, both of which are obtained by active probing. Although it can help RON distinguish paths with significant performance difference and select better alternate path, this estimate may not be accurate; a path with high score may actually have low available bandwidth [8]. In mTCP, the sender can monitor the performance of several paths in parallel. The throughput of each subflow provides a fairly good estimate of the available bandwidth on that path. This does not require any active probing because the data packets serve as probing packets. This can help mTCP discover and utilize better paths than the suboptimal *RON path* or direct path. We examined the paths in those 6 pairs and found that *MP₁* flows do take paths different from either the direct paths or the *RON paths*. The achieved throughput on those paths are higher than that of direct path or *RON path*.

Figure 14 shows the results of the remaining 9 pairs. By examining the paths, we find that all MP_1 flows degrade to single-path flows because of shared congestion, and the RON/INET/ MP_1 flows all take the same single path for the whole transfer. Hence, the throughput of MP_1 flows should be comparable to that of INET/RON flows, as shown in Figure 14. In three pairs, MP_1 flows obtain slightly lower performance than RON/INET flows, this is because different types of flows are run sequentially and there is minor fluctuations in the available bandwidth of a path over time.

6 Conclusions

In this paper, we present mTCP, a transport layer protocol, for improving end-to-end throughput and robustness. mTCP can efficiently aggregate the available bandwidth on several paths in parallel. To address the aggressiveness of mTCP during shared congestion, we integrate a shared congestion detection mechanism into our system so that correlated subflows can be suppressed. mTCP flows are more robust to path failures than TCP flows, because they will not stall even when some paths fail. The failure detection time is within several seconds. We also propose a heuristic to find disjoint paths based on traceroute. We have implemented our system on top of overlay networks and evaluated it on PlanetLab and Emulab.

References

- [1] <http://dast.nlanr.net/projects/iperf/>.
- [2] <http://www.emulab.net>.
- [3] <http://www.isi.edu/nsnam/ns>.
- [4] H. Adiseshu, G. M. Parulkar, and G. Varghese. A reliable and scalable striping protocol. In *Proceedings of ACM SIGCOMM*, 1996.
- [5] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of ACM SIGCOMM*, Aug. 2003.
- [6] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area Internet bottlenecks. In *Proceedings of ACM Internet measurement conference*, Oct. 2003.
- [7] M. Allman, H. Kruse, and S. Ostermann. An application-level solution to TCP's satellite inefficiencies. In *Proceedings of WOSBIS*, Nov. 1996.
- [8] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. *Proceedings of ACM SOSP*, Oct. 2001.
- [9] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee. On multiple description streaming with content delivery networks. In *Proceedings of IEEE INFOCOM*, 2002.
- [10] A. Banerjee. Simulation study of the capacity effects of dispersity routing for fault tolerant realtime channels. *Proceedings of ACM SIGCOMM*, Aug. 1996.
- [11] B. Chazelle. The discrepancy method: randomness and complexity. Cambridge University Press, 2000.
- [12] J. Duncanson. Inverse multiplexing. In *IEEE Communications Magazine*, volume 32, pages 34--41, 1994.
- [13] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: methodology and experience. *Proceedings of ACM SIGCOMM*, Aug. 2000.
- [14] T. Hacker, B. Athey, and B. Noble. The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network. In *Proc. of IPDPS*, 2002.
- [15] K. Harfoush, A. Bestavros, and J. Byers. Robust identification of shared losses using end-to-end unicast probes. *Proceedings of IEEE ICNP*, Oct. 2000.
- [16] H. Hsieh and R. Sivakumar. ptcp: An end-to-end transport layer protocol for striped connections. In *Proceedings of IEEE ICNP*, 2002.
- [17] H. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proceedings of ACM MOBICOM*, 2002.
- [18] D. Katabi, I. Bazzi, and X. Yang. An information theoretic approach for shared bottleneck inference based on end-to-end measurements. In *Class Project, MIT Laboratory for Computer Science*, 1999.
- [19] J. Lee, D. Gunter, B. Tierney, B. Allcock, J. Bester, J. Bresnahan, and S. Tuecke. Applied techniques for high bandwidth data transfers across wide area networks. In *Proceedings of CHEP*, Sept. 2001.
- [20] Y. Liang, E.G. Steinbach, and B. Girod. Real-time voice communication over the Internet using packet path diversity. In *Proceedings of ACM Multimedia*, 2001.
- [21] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *Proceedings of ICNP*, Nov. 2001.
- [22] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. *RFC 2018*, Oct. 1996.
- [23] A. Nakao, L. Peterson, and A. Bavier. A routing underlay for overlay networks. In *Proceedings of ACM SIGCOMM*, Aug. 2003.
- [24] T. Nguyen and A. Zakhori. Path diversity with forward error correction (pdf) system for packet switched networks. In *Proceedings of IEEE INFOCOM*, 2003.
- [25] V. Paxson. End-to-end routing behavior in the Internet. *Proceedings of ACM SIGCOMM*, Aug. 1996.
- [26] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *Proceedings of ACM HOTNET*, Oct. 2002.
- [27] D. Rubenstein, J. Kurose, and D. Towsley. Detecting shared congestion of flows via end-to-end measurement. *Proceedings of ACM SIGMETRICS*, June 2000.
- [28] S. Savage, A. Collins, and E. Hoffman. The end-to-end effects of Internet path selection. *Proceedings of ACM SIGCOMM*, Aug. 1999.
- [29] H. Sivakumar, S. Bailey, and R. L. Grossman. Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing*, 2000.
- [30] A. Snoeren. Adaptive inverse multiplexing for widearea wireless networks. In *Proc. of IEEE Conference on Global Communications*, 1999.
- [31] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with rocketfuel. *Proceedings of ACM SIGCOMM*, Aug. 2002.
- [32] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. In *RFC 2960*, Oct. 2000.
- [33] B. Traw and J. Smith. Striping within the network subsystem. In *IEEE Network*, volume 9, pages 22--32, 1995.
- [34] O. Younis and S. Fahmy. On efficient on-line grouping of flows with shared bottlenecks at loaded servers. In *Proceedings of IEEE ICNP*, Nov. 2002.
- [35] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. *Proceedings of ACM SIGCOMM*, Aug. 2002.
- [36] Y. Zhang, N. Duffield, V. Paxson, and S. Shenkar. On the constancy of Internet path properties. *Proceedings of Internet Measurement Workshop*, Nov. 2001.

Multihoming Performance Benefits: An Experimental Evaluation of Practical Enterprise Strategies

Aditya Akella, Srinivasan Seshan
Dept. of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Anees Shaikh
Network Software and Services
IBM T.J. Watson Research Center
Hawthorne, NY 10532

Abstract

Multihoming is increasingly being employed by large enterprises and data centers as a mechanism to extract good performance from their provider connections. Today, multihomed end-networks can employ a variety of commercial *route control* products to optimize performance over multiple ISP links. However, little is known about the mechanisms employed by such products and their relative trade-offs.

In this paper, we propose and evaluate a wide range practical schemes that could go into the design of a route control device and analyze their trade-offs. We implement the proposed schemes on a Linux-based Web proxy and perform a trace-based emulation of their relative performance benefits. We show that both passive and active monitoring based techniques are equally effective and could improve Web performance by about 25% when compared to using a single provider. Another key observation is that the conventional practice of employing historical measurement samples to monitor and predict ISP performance could, in fact, result in sub-optimal performance.

1 Introduction

Large enterprises, campuses, and data centers have traditionally used multihoming to multiple ISPs as a way of ensuring continued operation during connectivity outages or other ISP failures. While increased resilience and availability remain primary objectives of multihoming, there is increasing interest in deriving other benefits from multiple ISP connections. In particular, multihoming can be leveraged for improving wide-area network performance, lowering bandwidth costs, and optimizing the way in which upstream links are used [12].

A number of products provide these *route control* capabilities to large enterprise customers which have their own public AS number and advertise their IP address prefixes to upstream providers using BGP [20, 18, 10]. Recognizing that not all enterprises are large enough to warrant full BGP peering with upstream ISPs, another class of products extends

these advantages to smaller multihomed organizations which do not use BGP [14, 17, 7]. All of these products use a variety of mechanisms and policies for route control but aside from marketing statements, little is known about the relative quantitative benefits of these mechanisms.

In an recent measurement study to quantify the performance benefits from multihoming, it was shown that performance could potentially improve by more than 40% when multiple upstream providers are employed [4]. In that study, the focus was on the the maximum achievable benefits, assuming that the multihomed network had perfect information about the performance across all providers at any time and could change routes arbitrarily often. Hence, it is still unclear if, and how, these benefits can be realized in a more practical multihoming scenario.

In this paper we explore design alternatives to realize performance benefits from multihoming in practice, particularly for enterprises with multiple ISP connections. We focus primarily on mechanisms used for inbound route control, since enterprises are mainly interested in optimizing network performance for their own clients who download content from the Internet (i.e., sink data).

We evaluate a variety of active and passive measurement strategies for multihomed enterprises to estimate the instantaneous performance of their provider links and pick the best provider for a given transfer. These strategies are evaluated in the context of a NAT-based implementation to control the inbound ISP link used by enterprise connections. We address a number of practical issues such as the usefulness of past history to guide the choice of the best provider link, the effects of sampling frequency on measurement accuracy, and the overhead of managing performance information for a potentially very large set of target destinations. We evaluate these policies using several client workloads, and an emulated wide-area network where delay characteristics are based on a large set of real network delay measurements.

Our evaluation shows that active and passive measurement-based techniques are equally effective in extracting the performance benefits of using multiple providers, both offering

about 15-25% improvement when compared to using a single provider. We also show that the most current sample of the performance to a destination via a given provider is a reasonably good estimator of the near-term performance to the destination. We show that the overhead of collecting and managing performance information for various destinations is negligible. We also conduct an initial study of mechanisms to control the ISP link used by external Internet clients who initiate connections to servers hosted in the enterprise.

The rest of this paper is structured as follows. In Section 2, we describe our enterprise multihoming solution and the various strategies for estimating ISP performance and for route control. Section 3 describes our implementation in further detail. In Section 4, we discuss the experimental set-up and results from our evaluation of the solution. Section 5 discusses some limitations inherent to our approach. Related work is presented in Section 6. Finally, Section 7 summarizes the contributions of this paper.

2 Solution Overview

In order to realize the performance benefits of multihoming, a route control solution requires three key functions: (1) monitoring provider links, (2) choosing the best provider link at a given instant, and (3) directing traffic over the best provider links. Figure 1 illustrates each of the functions. We discuss the functional design of each of these below. We discuss the actual implementation details in Section 3.

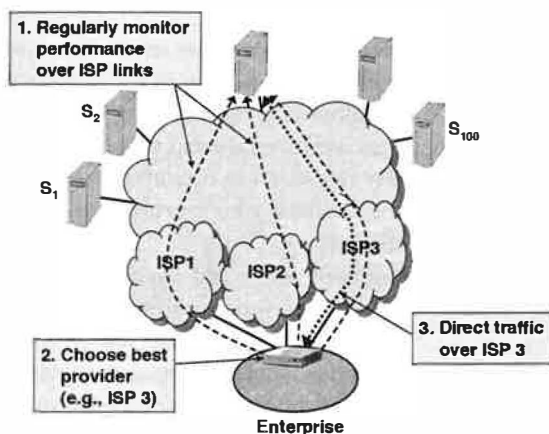


Figure 1: **Solution steps:** This figure illustrates the three main operations of an enterprise route control system.

2.1 Monitoring Provider Links

Selecting the right provider link over which to direct each transfer is crucial to realizing the performance benefits of multihoming from the enterprise network's perspective. The choice of the right ISP clearly depends on the time-varying performance of each provider link to each destination being accessed. However, network performance could vary over

small timescales, very drastically on some occasions [4, 22]. A multihomed enterprise, therefore, needs effective mechanisms to monitor the performance for most, if not all, destinations over each of its providers links.

There are two further issues in monitoring performance over provider links: *what* to monitor and *how*. In the enterprise case, one would ideally like to monitor the performance from every possible content provider over each ISP link. However, this may be infeasible in the case of a large enterprise which accesses content from many different sources. A simple solution to this problem is to monitor only the most important destinations on the basis of the volume of requests made from the enterprise (e.g., the top 100 most frequently accessed destinations). This would ensure that a significant fraction of all flows will experience good performance.

For the second question (i.e., how to monitor), two common approaches are active monitoring and passive monitoring. Active monitoring works by having the multihomed enterprise perform out-of-band measurements of performance to or from the destinations selected by the policy used to determine what to monitor. These measurements could be simple pings involving, for example, ICMP ECHO.REQUEST or TCP SYN packets to the destinations. These measurements are to be taken over each provider at regular intervals.

On the other hand, passive measurement mechanisms rely on observing the performance of ongoing transfers (i.e., in-band) to destinations, and using these observations as samples for estimating performance over the given provider. However, in order to ensure that there are enough samples over all providers, it may be necessary to *explicitly direct* some transfers over particular links.

An important component of monitoring performance is the *time interval* of monitoring. A long interval between performance samples implies using stale information to estimate provider performance. This might result in a suboptimal choice of the provider link for a particular destination. While using smaller time intervals would address this, it could have a negative impact as well. In active monitoring, frequent measurements inflate the out-of-band measurement traffic causing additional bandwidth and processing overhead; some destinations might interpret this traffic as a security threat. In passive monitoring, frequent sampling may cause too many connections to be directed over sub-optimal providers in an attempt to obtain performance samples. As such, a careful choice of the interval size is crucial.

2.2 Choosing the Best Provider

The next component is to select the best provider for a destination at a given time based on past measurement samples from monitoring provider links. The core issue here is whether, and how, historical data about ISP performance to a given destination should be used at all. The performance of an ISP link to a destination can be tracked by keeping a

smoothed, time-weighted estimate of the performance, for example an exponentially-weighted moving average (EWMA). If performance of using an ISP P to reach destination D at time t_i is s_{t_i} (as obtained from active or passive measurement) and the previous performance sample was from time t_{i-1} , then the EWMA metric at time t_i is:

$$EWMA_{t_i}(P, D) = (1 - e^{-(t_i - t_{i-1})/\alpha})s_{t_i} + e^{-(t_i - t_{i-1})/\alpha}EWMA_{t_{i-1}}(P, D)$$

where $\alpha > 0$ is a constant. A smaller value of α attaches less weight to historical samples. A value of $\alpha = 0$ implies no reliance on history. At any time, the provider with the best performance as calculated above could be chosen for a transfer. When no history is employed ($\alpha = 0$), only the most recent performance sample is used to evaluate the providers and select the best.

2.3 Directing Traffic Over Selected Providers

Once the best-performing provider for a transfer is identified, the traffic from the destination must be directed over the chosen link. This is the main inbound *route control* mechanism. Inbound control refers to selecting the right ISP or *incoming* interface on which to *receive* data. For an enterprise network, the primary mechanisms available are route advertisements and use of different addresses for different connections. Here, we discuss how these controls can be implemented.

If an enterprise has its own IP address block, it can advertise different address ranges to its upstream providers. Consider a site multihomed to two ISPs which owns a /19 address block. The site announces part of its address block on each provider link (e.g., a /20 sub-block on each link). Then, depending on which of the two provider links is considered superior for incoming traffic from a particular destination, the site would use a source address from the appropriate /20 address block. This ensures that all incoming packets for the connection would traverse the appropriate provider link. In cases where the enterprise is simply assigned an address block by its upstream provider, it may be necessary to also send outbound packets via the desired provider to ensure that the ISP forwards the packets.¹

The process of ensuring that a connection uses a particular address must be handled differently for connections that are initiated from the enterprise than for those that are accepted into the site from external clients, as discussed below.

Initiated Connections: Handling connections initiated from an enterprise site amounts to ensuring that the remote content provider transmits data such that the enterprise ultimately receives it over the chosen provider. Inbound control can be achieved by having the edge router translate the source

¹In fact, like most enterprise route control products, we enforce outbound route control by transmitting packets to a destination along the same provider as the one on which the traffic from the destination is received.

addresses on the connections initiated from its network to those belonging to the chosen provider's address block (i.e., the appropriate /20 block in the example above) via simple NAT-like mechanisms. This ensures that the replies from the destination will arrive over the appropriate provider.

Accepted Connections: Inbound route control over connections accepted into a site is necessary when the enterprise also hosts Internet servers which are accessed from outside. In this case, inbound control amounts to controlling the path (or the provider link) on which a given client is forced to send request and acknowledgment packets to the Web server. This is not easy since predicting client arrivals and forcing them to use the appropriate server address is generally not possible.

However, techniques based on DNS or deploying multiple versions of Web pages can help to achieve inbound control for externally initiated connection. For example, the enterprise can use a different version of a base Web page for each provider link. The hyperlinks for embedded objects in the page could be written with IP addresses corresponding to a given provider. Then, arriving clients would be given the appropriate base HTML page such that subsequent requests for the embedded objects arrive via the selected provider. On the other hand, the essential function of the DNS-based technique is to provide the address of the "appropriate" interface for each arriving client. A preliminary study of its effectiveness is discussed in Section 5. In this paper, we focus primarily on the case of enterprise-initiated connections.

3 Implementation Details

We implement the multihoming route control functions discussed above by extending a simple open source Web proxy called *TinyProxy* [3]. TinyProxy is a transparent, non-caching forward Web proxy that manages the performance of Web requests made by clients in a moderately-sized, multihomed enterprise. Below, we present the details of our implementation of the three basic multihoming components in TinyProxy. For the sake of simplicity, we assume that the proxy is being employed by a multihomed end-network with three ISP links.

3.1 Performance Monitoring Algorithms

We implement both the active and passive measurement mechanisms, described in Section 2.1, for monitoring the performance of upstream provider links.

3.1.1 Passive Measurement

The passive measurement module tracks the performance to destinations of interest by sampling provider links using Web requests initiated by clients in the enterprise. The basic strategy is to use new requests to sample an ISP's performance to a given destination if the performance estimate for that ISP

is older than the predefined sampling interval. If the module has current performance estimates for all links, then the connection is directed over the best link for the destination.

The module maintains a performance hash table keyed by the destination (i.e., either the IP address or the domain name of the destination). A hash table entry holds the current estimates of the performance to the destination via the three providers, along with an associated timestamp indicating the last time performance to the destination via the provider was measured. This is necessary for updating the EWMA estimate of performance (Section 2.2).

Notice that without some explicit control, the hash table maintains performance samples to all destinations, including those rarely accessed. One concern is that this could cause a high overhead of measurement, with connections to less popular destinations being all used up for obtaining performance samples. While maintaining explicit TTLs per entry might help flush out destinations that have not been accessed over a long period of time, it does not guarantee a manageable measurement overhead. Also, TTLs require maintaining a separate timer per entry, which is an additional overhead.

In view of this, we limit performance sampling to connections destined for the most popular sites, where popularity is measured in terms of aggregate client request counts, as follows: Hash entries also hold the number of *accesses* made to the corresponding destinations. Upon receiving a connection request for a given destination, we update the access count for the destination using an exponentially weighted moving average (EWMA). The EWMA weight is chosen so that the access count for the destination is reset to ~ 1 if it was not accessed for a long time, say 1 hour.

We use a hard threshold and monitor performance to destinations for which the total number of requests exceeds the threshold (by looking for live entries in the table with the access counts exceeding the threshold). In a naive hash table implementation for tracking the frequency counts of the various elements, identifying the popular destinations may take $O(\text{hash table size})$ time.

Other ways of tracking top destinations such as Iceberg Queries [8] or Sample-and-hold [6], may not incur such an overhead. Nevertheless, we stick with our approach for its simplicity of implementation. Also, as we will show later, the overhead from looking for the popular hash entries in our implementation is negligible. Note that this approach does not necessarily limit the actual number of popular destinations, for example in the relatively unlikely case that a very large number of destinations are accessed very often.

Figure 2 shows the basic operation of the passive monitoring scheme. When an enterprise client initiates a connection, the scheme first checks if the destination has a corresponding entry in the performance hash table (i.e., it is labeled popular). If not, the connection is simply relayed using a provider link chosen randomly, in a load-balancing fashion.

If there is an entry for the destination, the passive scheme

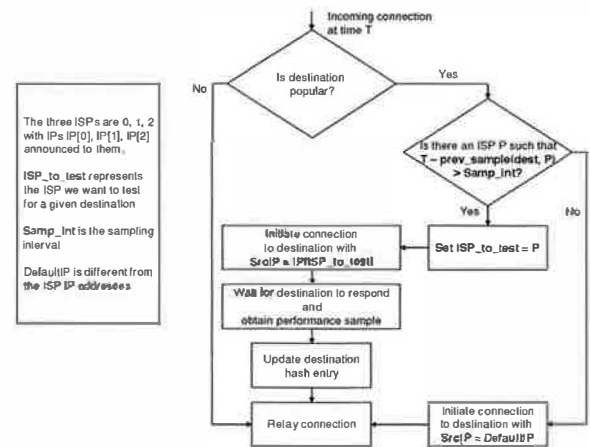


Figure 2: **Monitoring provider performance:** The passive measurement scheme.

scans the measurement timestamps for the three providers to see if the elapsed time since the last measurement on any of the links exceeds the predefined *sampling interval*. If so, the performance to the destination along one of these providers links is sampled using the current connection.

In order to obtain a measurement sample on a provider link, the scheme initiates a connection to the destination using a source IP address set such that the response will return via the link being sampled. Then, it measures the *turn-around time* for the connection, defined as the time between the transmission of the last byte of the client HTTP request, and the receipt of the first byte of the HTTP response from the destination. The observed turn-around time is used as the performance sample to the destination, and the corresponding entry in the hash table is updated using the EWMA method (Section 2.2). The remainder of the Web request proceeds normally, with the proxy relaying the data appropriately.

If all of the ISP links have current measurements (i.e., within the sampling interval), the proxy initiates a connection using the best link for the destination by setting the source IP address appropriately. We discuss these details in Section 3.3.

3.1.2 Active Measurement

Similar to passive measurement, the active measurement scheme also maintains a hash table of the performance estimates to candidate destinations over the three providers. For active measurement, we use two techniques to identify which destinations should be monitored.

FrequencyCounts. Just like the passive measurement mechanism, in this scheme we track the number of client requests directed to each destination. Every T seconds (the sampling interval), we initiate active probes to those destinations for which the number of requests exceeds a fixed threshold.

SlidingWindow. This scheme maintains a window of size C that contains the C most recently accessed destinations. The

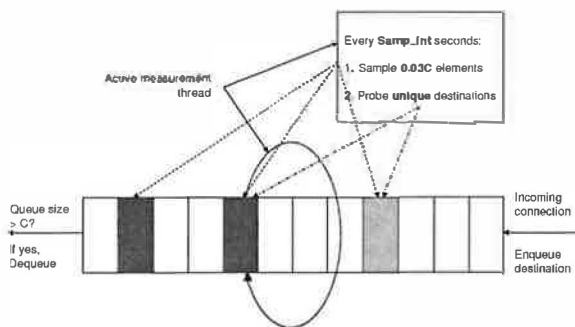


Figure 3: **Monitoring provider performance:** The *Sliding-Window* active measurement scheme.

window is implemented as a fixed size FIFO queue, in which destinations from newly initiated connections are inserted. If this causes the number of elements to exceed C , then the oldest in the window is removed. Every T seconds (the sampling interval), an active measurement thread scans the window and chooses $m\%$ of the elements at random. After discarding duplicate destinations from this subset, the active-measurement scheme measures the performance to the remaining destinations along the providers. This is illustrated in Figure 3.

The two active measurements schemes have their respective advantages and disadvantages. Notice that both the schemes effectively sample the performance to destinations that are accessed more often relative to others. However, there are a few key differences. First, *FrequencyCounts* is deterministic since it works with a reasonably precise set of the top destinations by popularity. *SlidingWindow*, on the other hand, may either miss a few popular destinations, or sample a few unpopular destinations. Second, *FrequencyCounts* in its simplest form, cannot easily track small, short-term shifts in the popularity of the destinations. These new, temporarily-popular destinations may not receive enough requests to exceed the threshold and force performance sampling for them, even though they are popular for a short time. *SlidingWindow* can effectively track small shifts in the underlying popularity distribution of the destinations and try to optimize performance to such temporarily popular destinations.

Probe operation. Once a destination is selected for active probing, the active measurement scheme sends three probes, with different source IP addresses, corresponding to the three providers and waits for the destination to respond. Since we found that a large fraction of popular Web sites filter ICMP ECHO_REQUEST packets, we employ a TCP-based probing mechanism. Specifically, we send a TCP SYN packet with the ACK bit set to port 80 and wait for an RST packet from the destination. We use the elapsed time as a sample of the turn-around time performance. We found that most sites respond promptly to the SYN+ACK packets.

When a response is received, we update the performance estimates to the destination for the corresponding provider, along with the measurement timestamp. As described above,

we update the performance estimate using the EWMA computation. If no response is received from a destination (which has an entry in the performance hash table), then a large positive value is used as the current measurement sample of the performance, and the performance is updated accordingly.

3.2 Switching Providers

After updating all provider entries for a destination in the performance hash, we switch to a new provider only if it offers at least a 10% performance improvement over the current best provider for the destination. Since the hash entries are updated at most once every T seconds (in either the passive or active measurement schemes), the choice of best provider per destination also changes at the same frequency.

3.3 NAT-based Inbound Route Control

Our inbound route control mechanism is based on manipulating NAT tables at the Web proxy to reflect the current choice of best provider. We use the `iptables` packet filtering facility in the Linux 2.4 kernel to install and update NAT tables at the proxy. The NAT rules associate destination addresses with the best provider link such that the source address on packets directed to a destination in the table are translated to an address that is announced to the chosen provider.

For example, suppose ISP 1 is selected for transfers involving destination 1.2.3.4 and the addresses 10.1.1.1 was announced over the link to ISP 1. Then we insert a NAT rule for the destination 1.2.3.4 that (1) matches packets with a source IP of `defaultIP` and destination 1.2.3.4, and (2) translates the source IP address on such packets to 10.1.1.1.

Notice that if the NAT rule blindly translates the source IP on all packets destined for 1.2.3.4 to 10.1.1.1, then it will not be possible to measure the performance to 1.2.3.4 via ISP 2, assuming that a different IP address, e.g., 10.1.1.2, was announced over the link to ISP 2. This is because the NAT translates the source address used for probing 1.2.3.4 across ISP 2 (i.e., 10.1.1.2) to 10.1.1.1, since ISP 1 is considered to be the best for destination 1.2.3.4. To get around this problem in our implementation, we simply construct the NAT rule to only translate packets with a specific source IP address (in this case `defaultIP`). Measurement packets that belong to probes (active measurement) or client connections (passive measurement) are sent with the appropriate source address, corresponding to the ISP to be measured.

4 Experimental Evaluation

In this section, we describe our experimental evaluation of the various design alternatives proposed in Section 3. These include the performance of passive versus active monitoring schemes, sensitivity to various measurement sampling intervals, and the overhead of managing performance information

for a large set of target destinations. We focus on understanding the benefits each scheme offers, including the set of parameters that result in the maximum advantage.

4.1 Experimental Set-up

We first describe our testbed setup and discuss how we emulate realistic wide-area network delays. Then we discuss key characteristics of the delay traces we employ in our emulation. Finally, we discuss the performance metrics we use to compare the proposed schemes.

4.1.1 Testbed topology

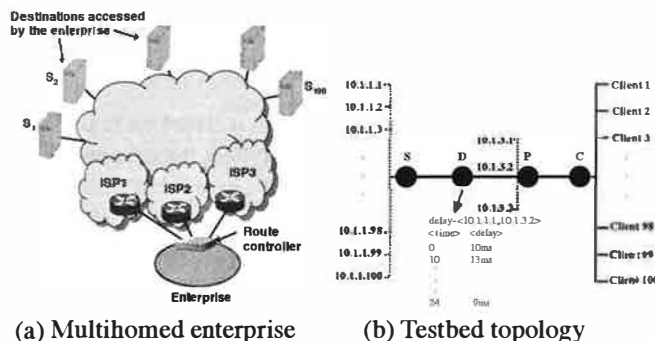


Figure 4: **Testbed topology:** The simple test-bed, shown in (b), is used to emulate the route control scenario shown in (a).

We use the simple testbed topology shown in Figure 4(b). Our goal is to emulate a moderately-sized enterprise with three provider connections and a client population of about 100 (shown in Figure 4(a)).

Node *S* in the topology runs a simple lightweight Web server and has one network interface configured with 100 different IP aliases – 10.1.1.1 through 10.1.1.100. Each alias represents an instance of a Web server – 10.1.1.1 being the most popular and 10.1.1.100 being the least popular.

Node *C* runs 100 instances of clients in parallel, each of which makes requests to the Web sites 10.1.1.1 through 10.1.1.100 as follows. The inter-arrival times between requests from a single client are Poisson-distributed with a mean of λ seconds. Notice that this mean inter-arrival rate translates into an average request rate of $\frac{100}{\lambda}$ requests per second at the server *S*. Each client request is for the i^{th} destination where i is sampled from the set $\{10.1.1.1, \dots, 10.1.1.100\}$ according to a Zipf distribution with an exponent ≈ 2 . In our evaluation, we set the parameters of the monitoring schemes (passive and active) so that the average rank of the destinations probed is 20, meaning that we explicitly track the top 40 most popular sites during each experiment. The object sizes requested by the client are drawn from a Pareto distribution with an exponent of 2 and a mean size of 5KB.

Node *P* in the topology runs the Web proxy (TinyProxy). It is configured with one “internal” interface on which the proxy

listens for connections from clients within the emulated enterprise. It has another interface with three IP aliases, 10.1.3.1, 10.1.3.2 and 10.1.3.3, each representing addresses announced over the three provider links.

Node *D* is a *delay element*, running WaspNet [13], a loadable kernel module providing emulation of wide-area network characteristics on the Linux platform. We modify WaspNet to enforce packet delays (along with drops, and bandwidth limits) on a per- \langle source IP, destination IP \rangle pair basis. We also modify it to support trace-based network delay emulation as illustrated in Figure 4(b).

In order to recreate realistic network delays between the clients and the servers in the testbed, we collect a set of wide area delay measurements using the Akamai content distribution network. We pick three Akamai server machines in Chicago, each attached to a unique provider. We then run pings at regular intervals of 10s from each of them to 100 other Akamai servers located in various US cities and attached to a variety of ISPs. The measurements were taken over a one-day period on Dec 7th, 2003.

In this measurement, the three Akamai machines in Chicago collectively act as a stand-in for a multihomed network with three provider connections. The 100 Akamai servers probed represent destinations contacted by end-nodes in the multihomed network. We use the series of delay samples between the three Akamai sources and the 100 destination servers as inputs to the WaspNet module to emulate delays across each provider link.

4.1.2 Compressing time

It is quite time-consuming to emulate the entire day’s worth of delays, multiple times over, to test and tune the different parameters in each scheme. One work-around could be to choose a smaller portion of the delay traces (e.g., 2 hours). However, a quick analysis of the delay traces we collected shows that there is not much variations in the delays along the probed paths on a 2-hour timescale. Since our goal is to understand how effective each scheme is over a wide range of operating conditions, it is important to test how well the schemes handle frequent changes in the performance of the underlying network paths. With this in mind, our approach is to compress the 24-hour delay traces by a factor of 10, to 2-hour delay traces and use these as the real inputs to the WaspNet delay module. In these 2-hour traces, performance changes in the underlying paths occur roughly 10 times more often when compared to the full 24-hour trace. The characteristics of the 2-hour delay traces collected from the nodes in Chicago are shown in Table 1, column 2. We use these delay traces in our emulation.

We also wanted to ensure that the delays observed from the Chicago source nodes were not significantly different from typical delays experienced by a well-connected, multihomed network located in a major U.S. metropolitan area. Hence,

we collected similar traces from 3 source nodes located in two other cities, namely New York and Los Angeles. These traces were collected on March 20th, 2004. The statistics for these latter traces are shown in columns 2 and 3 of Table 1. These statistics show that the Chicago-based traces we use in our experiments have roughly the same characteristics as those collected at the other metros.

	Chicago trace	NYC trace	LA trace
Mean time between performance changes	79s	101s	105s
Standard deviation of time between changes	337s	487s	423s
Mean extent of performance change	$\pm 33\%$	$\pm 28\%$	$\pm 34\%$
Standard deviation of extent of change	$\pm 26\%$	$\pm 22\%$	$\pm 27\%$
Mean time between performance changes of 30%	298s	261s	245s

Table 1: Characteristics of the delay traces. Here “performance” refers to the delay along a given path.

4.1.3 Comparison Metric

To evaluate the benefit from using various route control schemes we compare the response time of transfers when using the scheme (i.e., $Resp_{(x,scheme)}$, for a transfer x), with the response time when the best of the three providers is employed for each transfer ($\min_i\{Resp_{(x,ISP_i)}\}$):

$$\mathcal{R}_{scheme} = \frac{1}{|x|} \sum_x \frac{Resp_{(x,scheme)}}{\min_i\{Resp_{(x,ISP_i)}\}} \quad (1)$$

Where, $|x|$ is the total number of transfers. We call \mathcal{R} the “performance metric” or the “normalized response time”. The closer \mathcal{R} is to 1, the better the performance of the scheme.

In the above computation, the response times from employing the best provider for any transfer (the terms in the denominator above) are computed in an offline manner for each transfer by forcing it to use each of the three providers and selecting the provider offering the best response time.

4.2 Experimental Results

We perform our experiments on the Emulab [5] testbed. We use 600MHz Pentium III machines with 256MB RAM, running Red Hat 7.3. We first describe how we select different client workloads in our evaluation, and then move on to the evaluation of different route control strategies.

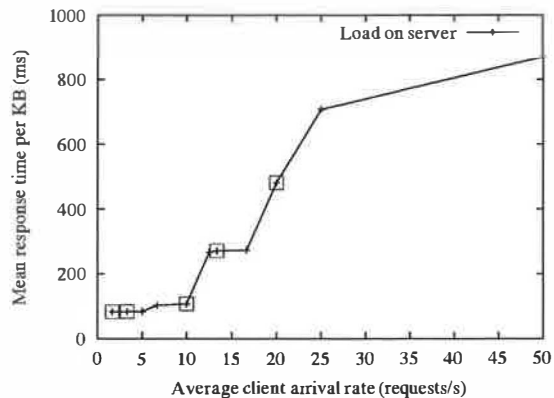


Figure 5: **Web server load profile:** Average response time in ms, per KB of the request, as a function of the average client arrival rate at the server in our topology (Figure 4(b)).

4.2.1 Selecting the Client Workloads

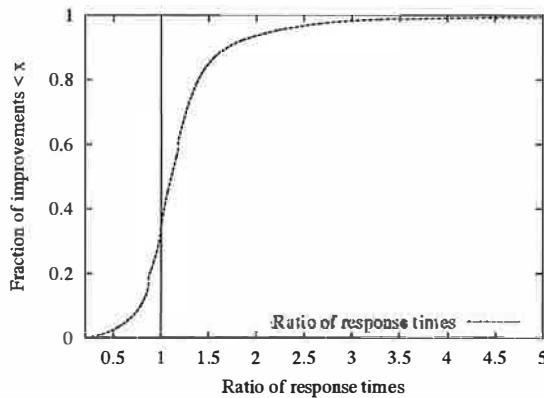
In Figure 5 we show the average response time per KB of client requests (i.e., the completion time for a request divided by the size of the request in KB), as a function of the average arrival rate of clients at the server S (i.e., $\frac{100}{\lambda}$ requests/s). The response time quickly degrades beyond an arrival rate of about 15 requests/s beyond which it increases only marginally with the request rate. We select five different points on this load curve (highlighted), corresponding to arrival rates of 1.7, 3.3, 10, 13.3 and 20 requests/s, and evaluate the proposed schemes under these workloads. These workloads represent various stress levels on the server S , while also ensuring that it is not overloaded. The high variability in response times in overload regimes might impact the confidence or accuracy of our comparison of the proposed schemes.

In the remainder of the evaluation we focus on addressing the following questions:

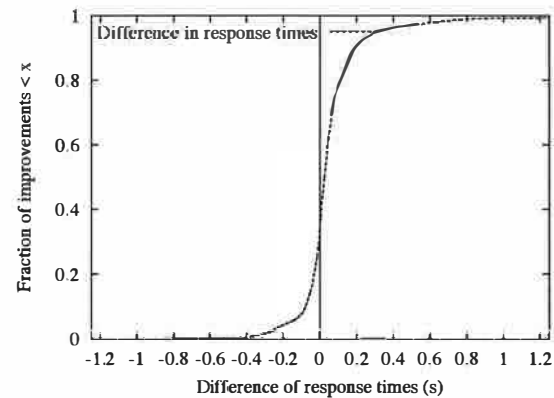
- To what extent do the route control schemes improve the performance of the multihomed site, relative to using the single best provider alone?
- Does employing historical samples help in better estimating future provider performance?
- How do active and passive measurement schemes compare in terms of the performance improvement they offer? Which of the two active measurement schemes – *SlidingWindow* or *FrequencyCounts* – works better?
- At what time intervals should samples for provider performance be collected?
- What overheads do the proposed mechanisms incur?

4.2.2 Improvements from Route Control

The aggregate performance improvement from the passive measurement-based schemes is shown in Figure 6. Here, we



(a) Ratio of response times



(b) Difference in response times

Figure 7: **Unrolling the averages:** Ratio and the difference in the response times from using just ISP 3 for all transfers relative to using the passive measurement scheme. The average client arrival rate in either case is 13.3 requests/s.

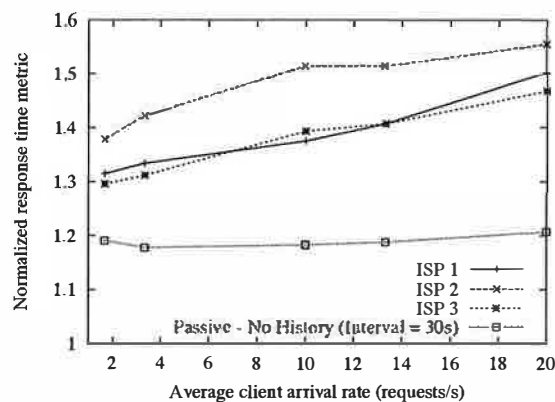


Figure 6: **Performance improvement:** The performance metric \mathcal{R} for the passive measurement scheme with EWMA parameter $\alpha = 0$ (no history employed) and sampling interval of 30s. The graph also shows the performance from the three individual providers.

set the EWMA parameter $\alpha = 0$ so that only the current measurement samples are used to estimate provider performance, and select a sampling interval of 30s. The figure plots the performance for the five client workloads. In addition, we show the performance from using the three providers individually.

The performance improvement relative to the best individual provider is significant – about 20-25% for the heavy workloads (right end of the graph) and about 10-15% for the light workloads (left end of the graph). The performance is still about 15-20% away from the optimal value of 1, however. The results for other sampling intervals (60s, 120s, 300s and 450s) are similar, and are omitted for brevity. The performance improvements from using the active measurement-based schemes are also similar and are discussed later.

Figures 7(a) and (b) illustrate the distribution of the response time improvements offered by the passive measurement scheme (for $\alpha = 0$ and sampling interval = 30s) relative

to being singly-home to the best provider from Figure 6, i.e., ISP 3. Figure (a) plots the CDF of the ratio of the response time from using ISP 3 to the response time from the passive measurement scheme across all transfers. These results are for the specific instance where the client arrival rate is 13.3 requests/s at the server. Figure (b) similarly plots the *difference* in the response times for the same client workload.

Notice, from either figure, that the passive measurement scheme improves the response time performance for over 65% of the transfers. Figure 7(a) shows that this route control scheme improves the response times by factors as large as 5 for a small fraction of transfers (about 1%), relative to being singly-homed. Similarly, Figure 7(b) shows that the scheme can improve the response time by more than 1s for some transfers. Notice also, from either figure, that the passive measurement-based scheme ends up offering sub-optimal performance for about 35% of the transfers.

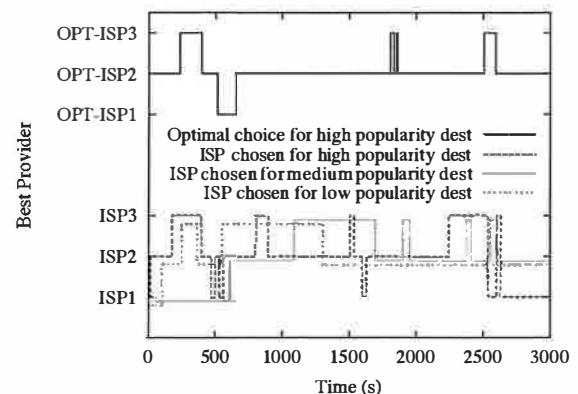


Figure 8: **Route control at work:** The providers chosen by the passive measurement-based route control scheme for destinations with different popularity levels.

Figure 8 illustrates the operation of the passive measurement-based scheme. In this figure, we show

the providers used over time for transfers to three different destinations – a popular destination (10.1.1.4), a moderately popular destination (10.1.1.16), and a less popular destination (10.1.1.38). Recall that the passive measurement-based scheme explicitly tracks and controls candidate paths to the 40 most popular destinations. The sampling interval is 30s and the client arrival rate is about 13.3 requests/s.

From this figure, we see that changes to the route for the popular destination is made every 160s on an average. For the moderate and less popular destinations, the intervals are 300s and 550s respectively. For the passive scheme, the number of route changes depends on the popularity of the destinations – the more popular a destination is, the higher the frequency of its route changes. Figure 8 also shows the optimal choice of providers for the popular destination as a function of time, as determined from the underlying delay traces. Comparing this with the ISPs actually selected by the scheme for this destination illustrates cases where the scheme sometimes makes a sub-optimal choice (e.g., between 750-800s, around 1500s, and 2250-2450s).

4.2.3 Employing History to Estimate Performance

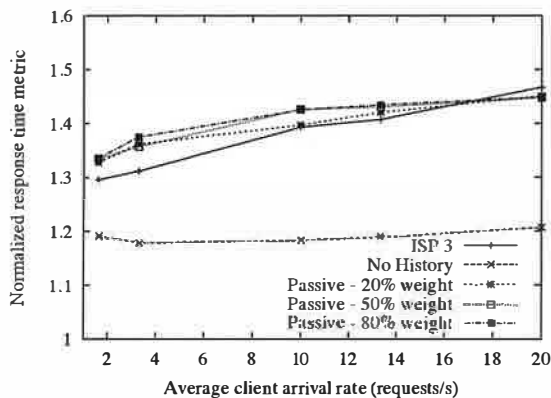


Figure 9: **Impact of history:** The performance achieved by relying on historical samples to varying degrees. These results are for the passive measurement-based strategy with a sampling interval of 30s.

Figure 9 plots the performance of the passive measurement scheme for three different values of the parameter α . These correspond to assigning 80%, 50% and 20% weight to the current measurement sample and the remaining weight to the past samples. Although we only show results for a sampling interval of 30s, the performance from other interval sizes are similar. The figure also plots the performance when no history is employed ($\alpha = 0$) and the performance from using ISP 3 alone. Notice that the performance from employing history is uniformly inferior in all situations, relative to employing no history. In fact, historical samples only serve to bring performance close to that from using the single best provider. These results show that the best way to estimate

provider performance is to just use the current performance sample as an estimate of near-term performance.

4.2.4 Active vs Passive Measurement

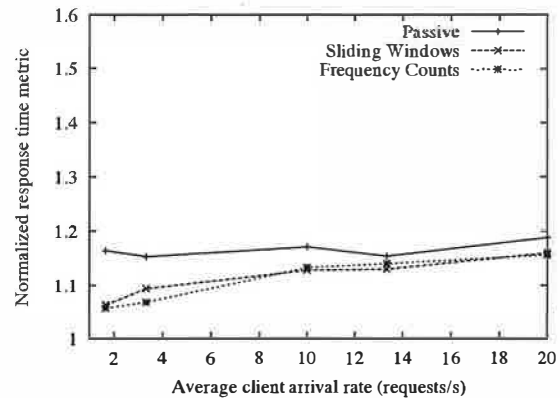


Figure 10: **Active vs passive measurement:** The performance of the two active measurement-based schemes, and the passive measurement scheme for a sampling interval of 120s.

In Figure 10, we compare the performance from the two active measurement based techniques (i.e., *SlidingWindow* and *FrequencyCounts*) with the passive measurement approach. Since our earlier results showed that history does not help in improving performance, henceforth we present results in which no history is employed. We compare the performance of the three measurement schemes for a common sampling interval of 120s across the five client workloads.

Note that the two active measurement schemes offer comparable performance. Unfortunately, the workloads we selected do not bring out other underlying trade-offs of these schemes (discussed earlier in Section 3.1.2). A detailed comparison of these active measurement schemes is future work.

Figure 10 also shows that the active measurement-based schemes offer slightly better performance than the passive measurement scheme: about 8-10% for the light workloads and 2-3% for the heavier workloads. This is expected, since the passive scheme uses existing transfers to obtain samples of performance across the, potentially sub-optimal, ISP links.

4.2.5 Frequency of Performance Monitoring

Figure 11 shows the impact of the measurement frequency on the aggregate performance for the passive measurement scheme (Figure 11(a)) and the *FrequencyCounts* active measurement scheme (Figure 11(b)). Each figure plots the results for the five client workloads.

From Figure 11(a) we notice that longer sampling intervals surprisingly offer slightly better performance for passive measurement. To understand this better, consider the curve for the client arrival rate of 10 requests/s. A client arrival

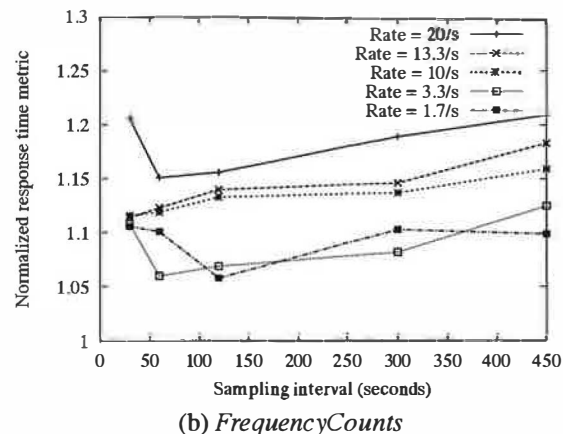
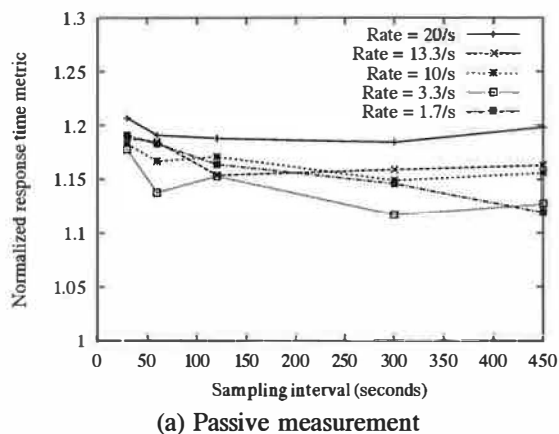


Figure 11: **Impact of the sampling interval:** The performance from using different sampling intervals from passive measurement-based and the *FrequencyCounts* active measurement-based schemes.

rate of 10 requests/s implies that an average of $10T$ connections are made by the clients every T seconds, where T is the sampling interval. However, in order to obtain samples for a fraction f of the 100 destinations over the three providers, the passive measurement scheme will have to force $300f$ connections across the provider links. This leaves a fraction $1 - \frac{30f}{T}$ which are not employed for measurement, and could be routed along the optimal provider, assuming that the passive measurement yields reasonably accurate estimate of performance². As T increases, the fraction of connections routed over the optimal path is likely to increase, resulting in a marginal improvement in performance. This explains the slight downward slopes in Figure 11(a).

At the same time, infrequent sampling (i.e., large values of T) can have a negative impact on the overall performance. This is not immediately clear from Figure 11(a). However, Figure 11(b), which plots the performance from the *FrequencyCounts* scheme as a function of the sampling interval, sheds more light on this effect. A sampling interval of 450s suffers a 5-8% performance penalty relative to a smaller interval such as 60s. Notice that in the case of *FrequencyCounts* too, aggressive sampling (e.g., an interval of 30s) could slightly impact overall performance on some occasions due to the increased software overheads at the proxy.

4.2.6 Analysis of overheads

As the performance results show, both passive and active measurement are still about 10-20% away from the optimal performance. Three key factors contribute to this gap: (1) the accuracy of measurement techniques, and correspondingly, the accuracy of provider choices, (2) overhead of performing measurement, and (3) software overhead, specifically, the

overhead of making frequent updates to the NAT table³ and employing NAT rules on a significant fraction of packets. In this section, we analyze the contribution of these factors on the eventual performance of the different schemes.

	Passive	Active FreqCount	Active SlidingWin
Total performance penalty	18%	14%	17%
Penalty from inaccurate estimation only	16%	12%	14%
Penalty from measurement and NAT only	2%	2%	3%

Table 2: **Analysis of performance overheads.** Here “penalty” is defined as the value of $\mathcal{R} - 1$ in each case.

Our approach to quantify the overhead of our implementation is to compare the performance derived from the choices made by the route control proxy, with the performance when the best ISP choices are made in an offline manner for each connection. Recall that in order to compute the performance metric \mathcal{R} , we evaluated the response time of each ISP for every transfer offline so that the best ISP link for each connection was known, independent of the route control mechanisms (the terms in the denominator in Equation 1). If we combine these offline response time values with the decisions made by the proxy, we can estimate the performance penalty due to incorrect choices, independent of the software overheads (i.e., #2 and #3 above). The difference between the resulting performance metric, \mathcal{R} , and 1 gives us the performance penalty, not including overheads of the implementation.

²About a third of the connections employed for measurement can be expected to be routed along their optimal providers

³We could allow routes to change less frequently than the sampling interval, T , (e.g., every $T' > T$ seconds) but since we do not use performance history, this would be equivalent to sampling and updating every T' seconds.

The penalties from the above analysis for the three proposed schemes are shown in Table 2, row 2. The client arrival rate is 13.3 requests/s and the sampling rate is 30s. In this table, the numbers in row 1 show the actual performance penalties suffered by the schemes in our implementation, taking all overheads into account (from Figure 11(a) and (b)). Notice that a large portion of the overall penalty is contributed by the inaccuracies in measurement and ISP selection (rows 1 and 2 are almost identical). Measurement and software overheads themselves result in a performance penalty of 2-3% (difference between rows 1 and 2, shown in row 3).

5 Implications and Discussion

The key findings from our evaluation are as follows:

1. The route control schemes we describe can significantly improve the performance of client transfers at a multi-homed site, up to 25% in our experiments.
2. We show that relying on historical samples to monitor performance of ISPs (e.g., using EWMA) is not very useful, and sometimes may be detrimental to performance. The most current measurement sample is a very good estimator of near-term performance of an ISP link.
3. Both passive and active measurement-based schemes offer competitive performance, with the latter offering better performance for lighter client workloads. For the generic Web workloads we tested with, both active measurement implementations – *SlidingWindow* and *FrequencyCounts* – showed similar performance benefits.
4. The overhead introduced by aggressive performance sampling may slightly reduce the overall performance benefit of route control schemes. A sampling interval on a minutes timescale, e.g., 60s, seems to offer very good performance overall.
5. The overhead from measurements and frequent updates to the NAT table are negligible. Most of the performance penalties arise from the inaccuracies of the measurement and estimation techniques.

5.1 Additional Issues

The route control mechanisms we presented and analyzed are a first attempt at understanding how to extract good performance from multiple provider connections in practice. There are clearly a number of ways in which they can be improved, however. Also, we do not address several important issues, such as ISP costs and the interplay of performance and reliability optimization. Below, we briefly discuss some of these potential improvements and issues.

Handling lost probes. In our implementation of the active probing schemes, we send just one probe when collecting a

performance sample for a (*ISP link, destination*) pair. It is therefore possible that lost probes, e.g., due to transient congestion or even timeouts, may be misinterpreted for poor performance of the provider path to the destination. This can in turn cause unwanted changes in the ISP choice for the destination. We can mitigate this by sending a short burst of, say, three probes per (*ISP link, destination*) pair. Then the performance reported by all three probes can be used to estimate the quality of the ISP link, perhaps with a weighting to account for any observed losses.

Hybrid passive and active measurements. The accuracy of passive measurement can be improved by sending active probes immediately after a failed passive probe, for example when the observed connection ends unexpectedly. This increases confidence that the failed connection is due to a problem with the provider link, as opposed to a transient effect.

In our implementation, paths to less popular destinations are not explicitly monitored (in both active and passive schemes). As a result, we may have to rely on passive observations of transfers to unpopular destination to ensure quick fail-over. For example, whenever the proxy observes a number of failures on connections to an unpopular destination, it can immediately switch the destination's default provider to one of the remaining two providers for future transfers.

Balancing performance and resilience. The goal of most current multihoming deployments is to provide resilient connectivity in the face of network failures. Hence, one of the main functions of a route control product is to respond quickly to ISP failures. One of our findings is that even with a relatively long sampling interval, the performance advantages of multihoming can be realized. A long interval can also slow the end-network's reaction to path failures, however. This can be addressed by sampling each destination with a sufficiently high frequency, while still keeping the probing overhead low. For example, a sampling interval of 60s with active measurement works well in such cases, providing reasonably low overhead and good performance (Figure 11(b)), while ensuring a failover time of about one minute.

ISP pricing structures. In our study, we ignore issues relating to the cost of the provider links. Different ISP connections may have very different pricing policies. One may charge a flat rate up to some committed rate, while another may use purely usage-based pricing or charge differently depending on whether the destination is "on-net" or "off-net." Though we do not consider how to optimize overall bandwidth costs, our evaluation of active and passive monitoring, and the utility of history, are central to more general schemes that optimize for both cost and performance.

Long-lived TCP flows. In our route control schemes, an update to a NAT entry for a destination in the midst of an ongoing transfer involving that destination could cause the transfer to fail (due to the change in source IP address). We did not observe many failed connections in our experiments,

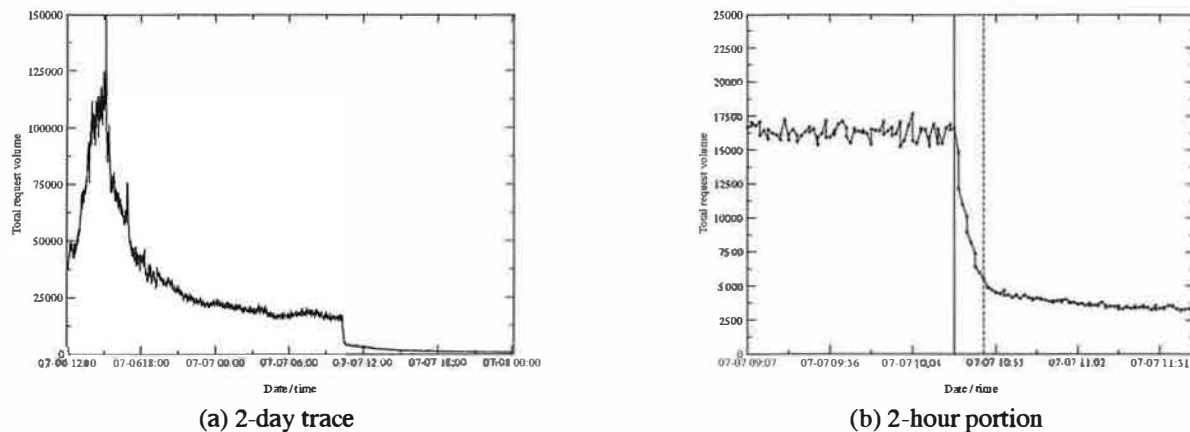


Figure 12: DNS responsiveness: This figure shows traffic volume over time just before and after a DNS change. The left graph (a) shows a 2-day period around the end of the event, while (b) focuses on a 2-hour period around the time of the DNS update.

however, and most of the flows were very short. However, this effect is nevertheless likely to have a pronounced impact on the performance of long-lived flows. It is possible to address this problem by delaying updates to NAT table until after ongoing large transfers complete. However, this increases the complexity of the implementation since it involves identifying flow lengths, and checking for the existence of other long-lived flows at the time of update. It may also force other short flows to the same destination to traverse sub-optimal ISPs while the NAT update is delayed.

Issues for further study. We do not address the impact that announcements of small address sub-blocks to different upstream ISPs (Section 2.3) has on the on the inflation of the routing table size in the core of the network. We also do not consider the potential impact of interactions when many enterprises deploy intelligent route control to each optimize their own multihomed connectivity. This will likely have an affect on the marginal benefits of the route control solutions themselves, and on the network as a whole. We leave these issues for future consideration.

Our implementation primarily considered handling connections initiated from within the enterprise, as these are common for current enterprise applications (e.g., to contact content providers). A route control product must also handle connections from outside clients, however, to enable optimized access to servers hosted in the enterprise network. Next, we describe some preliminary measurements regarding the usefulness DNS for externally-initiated connections.

5.2 DNS for Inbound Route Control

When deploying Internet servers in a multihomed environment, it is useful to be able to transparently direct connections initiated by external clients over a specific link, according to performance or other metrics. Recently, several route control device vendors have introduced features to use Domain Name System (DNS) resolution requests as a means to direct

inbound client traffic over the desired link. In this scheme, it is assumed that the destination IP address used by the client determines which ISP link is used for the connect request. Hence, by responding with the appropriate IP address when the client makes a request to resolve a service name (e.g., `www.service.com`), the inbound link can be selected. This is very similar to using DNS as a server selection mechanism in content distribution networks [19].

While DNS is a convenient and relatively transparent mechanism, it is unclear whether it can respond quickly enough for dynamic route control. Responses to name resolution requests have an associated time-to-live (TTL) value that determines how long the response should be cached by the client's local nameserver. Ideally, by setting the TTL to a very small value (e.g., 10s or even zero), it is possible to force external clients to resolve the IP address frequently, thus providing fast responsiveness. In practice, however, this is complicated by the behavior of the wide variety of applications and DNS servers deployed in the Internet. Many applications perform their own internal DNS caching that does not adhere to the expected behavior, and some older implementations of DNS software have been reported to ignore low TTL values. These artifacts make it difficult to predict how quickly clients will respond to changes communicated via DNS responses.

In order to quantify the responsiveness of DNS in practice, we perform a simple analysis of client behavior in response to DNS changes during a large Web event. We collect logs from a set of Web caches that served requests for content related to a Summer 2003 sporting event with global audience. During the event when the request rate was very high, the authoritative nameservers directed all clients to the set of caches with a 10min TTL. After the event, the nameservers were updated to direct clients to lower capacity origin servers. Ideally, all traffic to the caches should subside after 10min.

Figure 12 shows the aggregate request volume to all caches over time, just before and after the DNS change, where requests were gathered into 1-minute intervals. During the one-

hour period after the DNS change, requests came from about 13400 unique client IP addresses and 5600 unique IP subnets. The number of subnets is computed by clustering client IP addresses using BGP tables obtained from [11, 2].

Figure 12(a) shows the last part of the trace, with a clear peak occurring on the last day of the event, followed by a period of relatively constant and sustained traffic, and finally a sharp dropoff corresponding to the time when the DNS is updated. Figure 12(b) focuses on the time around the DNS update; the solid line denotes the time of the update and the dashed line is the time when the 10min TTL expires. Between these times, the request volume decreases by 66%. The remaining third of the traffic decays very slowly over a period of more than 12 hours. While this analysis is not definitive, it does suggest that DNS is at best a coarse-grained mechanism for controlling traffic.

6 Related Work

In a previous study we considered the extent to which multihoming can be leveraged by enterprises and Internet data centers to improve network performance [4]. This measurement-based study revealed the maximum benefits attainable in a variety of scenarios, but provided only limited guidance as to how to extract those improvements. In this paper, we perform an experimental evaluation of a number of practical techniques for using multihoming to improve performance.

In a study closely related to ours, the authors conduct a few trace-driven experiments to evaluate several design options using a commercial multihoming device [9, 17]. The evaluation focuses on the ability of several algorithms to balance load over multiple broadband-class links to provide service similar to a single higher-bandwidth link. The authors find that the effectiveness of hash-based link selection (i.e., hashing on packet header fields) in balancing load is comparable to load-based selection. In addition, their results show that managing load at a connection-level granularity is only slightly less effective than per-packet load balancing. They also show that using knowledge of the asymmetric nature of some applications (e.g., Web connections) can be useful in improving traffic balance, although it requires additional application-specific information.

A number of vendors have recently developed dedicated networking appliances [15, 7, 14] or software stacks [21, 16] for optimizing the use of multihomed connectivity in enterprises settings where BGP is not used. Most of these products use techniques similar to those we evaluate in our study, though their focus is geared more toward balancing load and managing bandwidth costs across multiple ISP links, rather than optimizing performance. All of these use NAT-based control of inbound traffic and DNS to influence links used by external client-initiated connections. They also ensure, by tracking sessions or using policy-based routing, that the same ISP link is used in both directions.

Another class of products and services are targeted at settings where BGP is employed, for examples large data centers or campuses [18, 1]. These products mainly focus on outbound control of routes and, as such, are more suited for content providers which primarily source data. Details of the algorithms used by any of the above commercial products to monitor link performance or availability are generally proprietary, and little information is available on specific mechanisms or parameter settings. Here, we review the general approaches taken in enterprise route control products.

Most commercial products employ both ICMP ping and TCP active probes to continuously monitor the health of upstream links, enabling rapid response to failure. In some cases, hybrid passive and active monitoring is used to track link performance. For example, when a connection to a previously unseen destination is initiated from an enterprise client, active probes across the candidate links sample performance to the destination. Connections to known destinations, on the other hand, are monitored passively to update performance samples. Another approach is to use active probing for monitoring link availability, and passive monitoring for performance sampling. Some products also allow static rules to dictate which link to use to reach known destinations networks.

Finally, some products use “race”-based performance measurements, in which SYN packets sent by enterprise clients to initiate connections are replicated by the route control device on all upstream ISPs (using source NAT). The link on which the corresponding SYN-ACK arrives from the server is used for the remainder of the connection. The route control device sends RST packets along the slower paths so that the server can terminate the in-progress connection establishment state. The choice of best link is cached for some time so that subsequent connections that arrive within a short time period need not trigger a new race unless a link failure is detected.

7 Summary and Ongoing Work

Our goal in this paper was to quantitatively evaluate a variety of practical mechanisms and policies for realizing performance benefits from ISP multihoming. We focused on the scenario of multihomed enterprises that wish to leverage multiple providers to improve the response time performance for clients who download content from Internet Web servers. Using a real Linux-based route control implementation and an emulated wide-area network testbed, we experimentally evaluated several design alternatives. These included the performance of passive versus active monitoring schemes, sensitivity to various measurement sampling intervals, and techniques to manage performance information for a potentially very large set of target destinations.

Our evaluation shows that both active and passive measurement-based route control schemes offer significant performance benefits in practice, between 15% and 25%, when compared with using the single best-performing ISP

provider. Our experiments also show that using historical performance to choose the best ISP link is not necessary – the most current measurement sample gives a good estimate. We showed that the performance penalty from collecting and managing performance data across various destinations is negligible.

Although our evaluation was done using an emulated wide-area network and actual delay traces, it is valuable to deploy our implementation in a multihomed site for further experimentation and evaluation. To this end, we are planning to install our route control proxy device in a commercial multihomed data center in which we can perform additional experiments and uncover other wide-area effects.

Acknowledgments

We are grateful to Prof. Bruce Maggs (CMU) for his support and assistance in facilitating measurement experiments on the Akamai network. We are grateful to Herbie Pearthree, Rance Smith (IBM Global Services), Jehan Sanmugaraja and Paul Dantzig (IBM Research), for their assistance in collecting data for the DNS analysis and discussions regarding the practical usage of route control and DNS. Our work has greatly benefited from discussions with Ashwin Bharambe, Dave Maltz, Hui Zhang (CMU), Dave Andersen (MIT) and Sridhar Machiraju (UC Berkeley). Finally, we thank our shepherd, Carl Staelin, and our anonymous reviewers for their invaluable feedback on the presentation of this paper.

References

- [1] Sockeye Networks, Inc. <http://www.sockeye.com>.
- [2] University of Oregon, RouteViews Project. <http://www.routeviews.org>.
- [3] Tinyproxy: A Simple Light-weight Web Proxy. <http://tinyproxy.sourceforge.net>, November 2003.
- [4] AKELLA, A., MAGGS, B., SESHAN, S., SHAIKH, A., AND SITARAMAN, R. A Measurement-Based Analysis of Multihoming. In *Proc. ACM SIGCOMM 2003* (Karlruhe, Germany, August 2003).
- [5] Emulab: Network Emulation Testbed. <http://www.emulab.net/>.
- [6] ESTAN, C., AND VARGHESE, G. New Directions in Traffic Measurement and Accounting. In *Proc. ACM SIGCOMM 2002* (Pittsburgh, PA, August 2002).
- [7] F5 Networks: BIG-IP Link Controller. http://www.f5.com/f5products/pdfs/SS_BIG-IP_LC.pdf, 2003.
- [8] FANG, M., SHIVAKUMAR, N., GARCIA-MOLINA, H., MOTWANI, R., AND ULLMAN, J. D. Computing Iceberg Queries Efficiently. In *Proc. VLDB 1998* (New York, August 1998).
- [9] GUO, F., CHEN, J., LI, W., AND CKER CHIUH, T. Experiences in Building a Multihoming Load Balancing System. In *INFOCOM 2004* (Hong Kong, March 2004).
- [10] Internap Network Services: Flow Control Platform. <http://www.internap.com>.
- [11] KRISHNAMURTHY, B., AND WANG, J. On Network-Aware Clustering of Web Clients. In *Proc. ACM SIGCOMM 2000* (Stockholm, Sweden, August 2000).
- [12] MORISSEY, P. Route Optimizers: Mapping Out the Best Route. *Network Computing* (December 2003). <http://www.nwc.com/showitem.jhtml?docid=1425f2>.
- [13] NAHUM, E. M., ROSU, M., SESHAN, S., AND ALMEIDA, J. The Effects of Wide-Area Conditions on WWW Server Performance. In *Proc. ACM SIGMETRICS* (Cambridge, MA, June 2001).
- [14] Nortel Networks: Alteon Link Optimizer. <http://www.nortelnetworks.com/products/01/alteon/optimizer/collateral/n%20102801-010203.pdf>, 2003.
- [15] Radware: Peer Director. <http://www.radware.com/content/products/pd/>.
- [16] Rainfinity: Overview of RainConnect. http://www.rainfinity.com/products/wp_rc_overview.pdf, 2004.
- [17] Rether Networks: Internet Service Management Device. <http://rether.com/ISMD.htm>.
- [18] RouteScience Technologies: PathControl. <http://www.routescience.com/products>.
- [19] SHAIKH, A., TEWARI, R., AND AGRAWAL, M. On the Effectiveness of DNS-based Server Selection. In *Proc. IEEE INFOCOM* (Anchorage, AK, April 2001).
- [20] STEWART, J. W. *BGP4: Inter-Domain Routing in the Internet*. Addison-Wesley, 1999.
- [21] Stonesoft: Multi-Link Technology. http://www.stonesoft.com/files/products/StoneGate/SG_Multi-Link_Technology_Whitepaper.pdf, October 2001.
- [22] ZHANG, Y., DUFFIELD, N., PAXSON, V., AND SHENKER, S. On the Constancy of Internet Path Properties. In *Proc. ACM SIGCOMM Internet Measurement Workshop (IMW)* (November 2001).

Handling Churn in a DHT

Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz
University of California, Berkeley and Intel Research, Berkeley
{srhea,geels,kubitron}@cs.berkeley.edu, troscoe@intel-research.net

Abstract

This paper addresses the problem of *churn*—the continuous process of node arrival and departure—in distributed hash tables (DHTs). We argue that DHTs should perform lookups quickly and consistently under churn rates at least as high as those observed in deployed P2P systems such as Kazaa. We then show through experiments on an emulated network that current DHT implementations cannot handle such churn rates. Next, we identify and explore three factors affecting DHT performance under churn: reactive versus periodic failure recovery, message timeout calculation, and proximity neighbor selection. We work in the context of a mature DHT implementation called *Bamboo*, using the ModelNet network emulator, which models in-network queuing, cross-traffic, and packet loss. These factors are typically missing in earlier simulation-based DHT studies, and we show that careful attention to them in Bamboo's design allows it to function effectively at churn rates at or higher than that observed in P2P file-sharing applications, while using lower maintenance bandwidth than other DHT implementations.

1 Introduction

The popularity of widely-deployed file-sharing services has recently motivated considerable research into peer-to-peer systems. Along one line, this research has focused on the design of better peer-to-peer algorithms, especially in the area of structured peer-to-peer overlay networks or distributed hash tables (e.g. [20, 22, 24, 27, 30]), which we will simply call DHTs. These systems map a large identifier space onto the set of nodes in the system in a deterministic and distributed fashion, a function we alternately call *routing* or *lookup*. DHTs generally perform these lookups using only $O(\log N)$ overlay hops in a network of N nodes where every node maintains only $O(\log N)$ neighbor links, although recent research has explored the tradeoffs in storing more or less state.

A second line of research into P2P systems has focused on observing deployed networks (e.g. [5, 9, 13, 25]). A significant result of this research is that such networks are characterized by a high degree of churn. One metric of

churn is node *session time*: the time between when a node joins the network until the next time it leaves. Median session times observed in deployed networks range from as long as an hour to as short as a few minutes.

In this paper we explore the performance of DHTs in such dynamic environments. DHTs may be better able to locate rare files than existing unstructured peer-to-peer networks [18]. Moreover, it is not hard to imagine that other proposed uses for DHTs will show similar churn rates to file-sharing networks—application-level multicast of a low-budget radio stream, for example. In spite of this promise, we show that short session times cause a variety of negative effects on two mature DHT implementations we tested. Both systems exhibit dramatic latency growth when subjected to increasing churn, and in one implementation the network eventually partitions, causing subsequent lookups to return inconsistent results. The remainder of this paper is dedicated to determining whether a DHT can be built such that it continues to perform well as churn rates increase.

We demonstrate that DHTs can in fact handle high churn rates, and we identify and explore several factors that affect the behavior of DHTs under churn. The three most important factors we identify are:

- reactive versus periodic recovery from failures
- calculation of message timeouts during lookups
- choice of nearby over distant neighbors

By *reactive recovery*, we mean the strategy whereby a DHT node tries to find a replacement neighbor immediately upon noticing that an existing neighbor has failed. We show that under bandwidth-limited conditions, reactive recovery can lead to a positive feedback cycle that overloads the network, causing lookups to have high latency or to return inconsistent results. In contrast, a DHT node may recover from neighbor failure at a fixed, periodic rate. We show that this strategy improves performance under churn by allowing the system to avoid positive feedback cycles.

The manner in which a DHT chooses timeout values during lookups can also greatly affect its performance under churn. If a node performing a lookup sends a message

to a node that has left the network, it must eventually timeout the request and try another neighbor. We demonstrate that such timeouts are a significant component of lookup latency under churn, and we explore several methods of computing good timeout values, including virtual coordinate schemes as used in the Chord DHT.

Finally, we consider *proximity neighbor selection* (PNS), where a DHT node with a choice of neighbors tries to select those that are most nearby itself in network latency. We compare several algorithms for discovering nearby neighbors—including algorithms similar to those used in the Chord, Pastry, and Tapestry DHTs—to show the tradeoffs they offer between latency reduction and added bandwidth.

We have augmented the Bamboo DHT [23] such that it can be configured to use any of the design choices described above. As such, we can examine each design decision independently of the others. Moreover, we examine the performance of each configuration by running it on a large cluster with an emulated wide-area network. This methodology is particularly important with regard to the choice of reactive versus periodic recovery as described above. Existing studies of churn in DHTs (e.g. [7, 8, 16, 19]) have used simulations that—unlike our emulated network—did not model the effects of network queuing, cross traffic, or message loss. In our experience, these effects are primary factors contributing to DHTs' inability to handle churn. Moreover, our measurements are conducted on an isolated network, where the only sources of queuing, cross traffic, and loss are the DHTs themselves; in the presence of heavy background traffic, we expect that such network realities will exacerbate the ability of DHTs to handle even lower levels of churn.

Of course, this study has limitations. Building and testing a complete DHT implementation on an emulated network is a major effort. Consequently, we have limited ourselves to studying a single DHT on a single network topology using a relatively simple churn model. Furthermore, we have not yet studied the effects of some implementation decisions that might affect the performance of a DHT under churn, including the use of alternate routing table neighbors as in Kademlia and Tapestry, or the use of iterative versus recursive routing. Nevertheless, we believe that the effects of the factors we have studied are dramatic enough to present them as an important early study in the effort to build a DHT that successfully handles churn.

The rest of this paper is structured as follows: in the next section we review how DHTs perform routing or lookup, with particular reference to Pastry, whose routing algorithm Bamboo also uses. In Section 3, we review existing studies of churn in deployed file-sharing networks, describe the way we model such churn in our emulated network, and quantify the performance of mature DHT

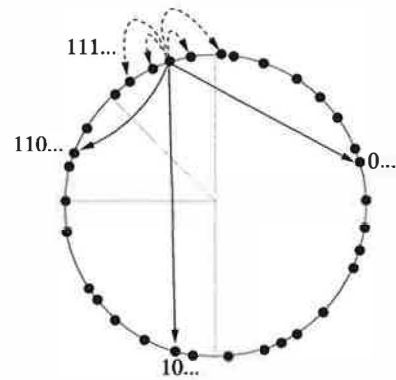


Figure 1: *Neighbors in Pastry and Bamboo.* A node's neighbors are divided into its *leaf set*, shown as dashed arrows, and its *routing table*, shown as solid arrows.

implementations under such churn. In Section 4, we study each of the factors listed above in isolation, and describe how Bamboo uses these techniques. In Section 5, we survey related work, and in Section 6 we discuss important future work. We conclude in Section 7.

2 Introduction to DHT Routing

In this section we present a brief review of DHT routing, using Pastry [24] as an example. The *geometry* and *routing algorithm* of Bamboo are identical to Pastry; the difference (and the main contribution of this paper) lies in how Bamboo maintains the geometry as nodes join and leave the network and the network conditions vary.

DHTs are structured graphs, and we use the term *geometry* to mean the pattern of neighbor links in the overlay network, independent of the routing algorithms or state management algorithms used [12].

Each node in Pastry is assigned a numeric identifier in $[0, 2^{160})$, derived either from the SHA-1 hash of the IP address and port on which the node receives packets or from the SHA-1 hash of a public key. As such, they are well-distributed throughout the identifier space.

In Pastry, a node maintains two sets of neighbors, the *leaf set* and the *routing table* (see Figure 1). A node's leaf set is the set of $2k$ nodes immediately preceding and following it in the circular identifier space. We denote this set by L , and we use the notation L_i with $-k \leq i \leq k$ to denote the members of L , where L_0 is the node itself.

In contrast, the routing table is a set of nodes whose identifiers share successively longer prefixes with the source node's identifier. Treating each identifier as a sequence of digits of base 2^b and denoting the routing table entry at row l and column i by $R_l[i]$, a node chooses its neighbors such that the entry at $R_l[i]$ is a node whose identifier matches its own in exactly l digits and whose

```

if ( $L_{-k} \leq D \leq L_k$ )
    next_hop =  $L_i$  s.t.  $|D - L_i|$  is minimal
else if ( $R_l[D[l]] \neq \text{null}$ )
    next_hop =  $R_l[D[l]]$ 
else
    next_hop =  $L_i$  s.t.  $|D - L_i|$  is minimal

```

Figure 2: *The Bamboo routing algorithm.* The code shown chooses the next routing hop in for a message with destination D , where D matches the identifier of the local node in the first l digits.

$(l + 1)$ th digit is i . In the experiments in this paper, Bamboo uses binary digits ($b = 1$), though it can be configured to use any base.

The basic operation of a DHT is to consistently map identifiers onto nodes from any point in the system, a function we call *routing* or *lookup*. Pastry achieves consistent lookups by directing each identifier to the node with the numerically closest identifier. Algorithmically, routing proceeds as shown in Figure 2. To route a message with key D , a node first checks whether D lies within its leaf set, and if so, forwards it to the numerically closest member of that set (modulo 2^{160}). If that member is the local node, routing terminates. If D does not fall within the leaf set, the node computes the length l of the longest matching prefix between D and its own identifier. Let $D[i]$ denote the i th digit of D . If $R_l[D[l]]$ is not empty, the message is forwarded on to that node. If neither of these conditions is true, the message is forwarded to the member of the node's leaf set numerically closest to D . Once the destination node is reached, it sends a message back to the originating node with its identifier and network address, and the lookup is complete.

We note that a node can often choose between many different neighbors for a given entry in its routing table. For example, a node whose identifier begins with a 1 needs a neighbor whose identifier begins with a 0, but such nodes make up roughly half of the total network. In such situations, a node can choose between the possible candidates based on some metric. *Proximity neighbor selection* is the term used to indicate that nodes in a DHT use network latency as the metric by which to choose between neighbor candidates.

Using this design, Pastry and Bamboo perform lookups in $O(\log N)$ hops [24], while the leaf set allows forward progress (in exchange for potentially longer paths) in the case that the routing table is incomplete. Moreover, the leaf set adds a great deal of *static resilience* to the geometry; Gummadi et al. [12] show that with a leaf set of 16 nodes, even after a random 30% of the links are broken there are still connected paths between all node pairs in a

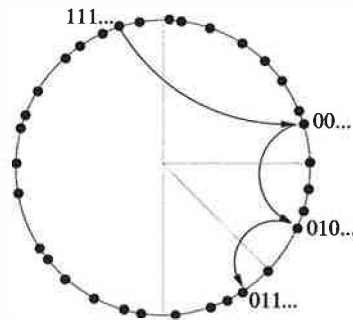


Figure 3: *Recursive lookup.* To find the node closest to identifier 011, the node whose identifier starts with 111 sends a lookup message to its neighbor whose first digit is 0. This node then forwards the query to its neighbor whose first two digits are 01, and from there the node is forwarded to the neighbor whose first three digits are 011.

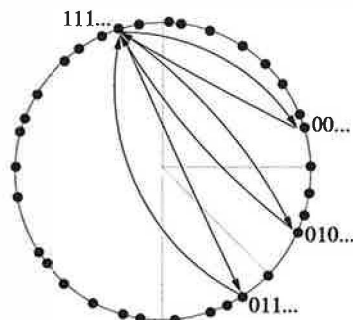


Figure 4: *Iterative lookup.* An iterative lookup involves the same nodes as a recursive one, but instead of forwarding the message, each intermediate node responds to the source with the address of the next hop.

network of 65,536 nodes. This resilience is important in handling failures in general and churn in particular, and was the reason we chose the Pastry geometry for use in Bamboo. We could also have used a pure ring geometry as in Chord, extending it to account for proximity in neighbor selection as described in [12].

The manner in which we have described routing so far is commonly called *recursive* routing (Figure 3). In contrast, lookups may also be performed *iteratively*. As shown in Figure 4, an iterative lookup involves the same nodes as a recursive one, but the entire process is controlled by the source of the lookup. Rather than asking a neighbor to forward the lookup through the network on its behalf, the source asks that neighbor for the network address of the next hop. The source then asks the newly-discovered node the same question, repeating the process until no further progress can be made, at which point the lookup is complete.

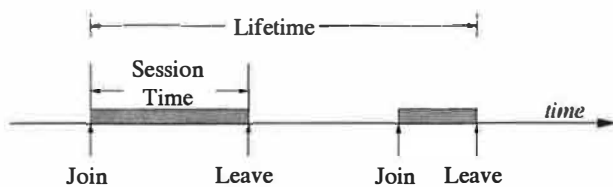


Figure 5: *Metrics of churn.* With respect to the routing and lookup functionality of a DHT, the *session times* of nodes are more relevant than their *lifetimes*.

3 The Problem of Churn

There have been very few large-scale DHT-based application deployments to date, and so it is hard to derive good requirements on churn-resilience. However, P2P file-sharing networks provide a useful starting point. These systems provide a simple indexing service for locating files on those peer nodes currently connected to the network, a function which can be naturally mapped onto a DHT-based mechanism. For example, the Overnet file-sharing system uses the Kademlia DHT to store such an index. While some DHT applications (such as file storage as in CFS [10]) might require greater client availability, others may show similar churn rates to file-sharing networks (such as end-system multicast or a rendezvous service for instant messaging). As such, we believe that DHTs should at least handle churn rates observed in today's file-sharing networks. To that end, in this section we survey existing studies of churn in deployed file-sharing networks, describe the way we model such churn in our emulated network, and quantify the performance of mature DHT implementations under such churn.

Studies of existing file-sharing systems mainly use two metrics of churn (see Figure 5). A node's *session time* is the elapsed time between it joining the network and subsequently leaving it. In contrast, a node's *lifetime* is the time between it entering the network for the first time and leaving the network permanently. The sum of a node's session times divided by its lifetime is often called its *availability*. One representative study [5] observed median session times on the order of tens of minutes, median lifetimes on the order of days, and median availability of around 30%.

With respect to the lookup functionality of a DHT, we argue that session time is the most important metric. Even temporary loss of a routing neighbor weakens the correctness and performance guarantees of a DHT, and unavailable neighbors reduce a node's effective connectivity, forcing it to choose suboptimal routes and increasing the destructive potential of future failures. Since nodes are often unavailable for long periods, remembering neighbors that have failed is of little value in performing lookups. While remembering neighbors is useful for applications like storage [6], it is of little value for *lookup* operations.

First Author	Systems Observed	Session Time
Saroiu [25]	Gnutella, Napster	50% \leq 60 min.
Chu [9]	Gnutella, Napster	31% \leq 10 min.
Sen [26]	FastTrack	50% \leq 1 min.
Bhagwan [5]	Overnet	50% \leq 60 min.
Gummadi [13]	Kazaa	50% \leq 2.4 min.

Table 1: *Observed session times in various peer-to-peer systems.* The median session time ranges from an hour to a minute.

3.1 Empirical studies

Elsewhere [23] we have surveyed published studies of deployed file-sharing networks. Table 1 shows a summary of observed session times. At first sight, some of these values are surprising, and may be due to methodological problems with the study in question or malfunctioning of the system under observation. However, it is easy to image a user joining the network, downloading a single file (or failing to find it), and leaving, making session times of a few minutes at least plausible. To be conservative, then, we would like a DHT to be robust for median session times from as much as an hour to as little as a minute.

3.2 Experimental Methodology

Our platform for measuring DHT performance under churn is a cluster of 40 IBM xSeries PCs, each with Dual 1GHz Pentium III processors and 1.5GB RAM, connected by Gigabit Ethernet, and running either Debian GNU/Linux or FreeBSD. We use ModelNet [28] to impose wide-area delay and bandwidth restrictions, and the Inet topology generator [3] to create a 10,000-node wide-area AS-level network with 500 client nodes connected to 250 distinct stubs by 1 Mbps links. To increase the scale of the experiments without overburdening the capacity of ModelNet by running more client nodes, each client node runs two DHT instances, for a total of 1,000 DHT nodes.

Our control software uses a set of wrappers which communicate locally with each DHT instance to send requests and record responses. Running 1000 DHT instances on this cluster (12.5 nodes/CPU) produces CPU loads below one, except during the highest churn rates. Ideally, we would measure larger networks, but 1000-node systems already demonstrate problems that will surely affect larger ones.

In an experiment, we first bring up a network of 1000 nodes, one every 1.5 seconds, each with a randomly assigned gateway node to distribute the load of bootstrapping newcomers. We then churn nodes until the system performance levels out; this phase normally lasts 20-30 minutes but can take an hour or more. Node deaths are timed by a Poisson process and are therefore uncorrelated and bursty. A new node is started each time one is killed,

maintaining the total network size at 1000. This model of churn is similar to that described by Liben-Nowell et al [17]. In a Poisson process, an event rate λ corresponds to a median inter-event period of $\ln 2/\lambda$. For each event we select a node to die uniformly at random, so each node's session time is expected to span N events, where N is the network size. Therefore a churn rate of λ corresponds to a median node session time of

$$t_{\text{med}} = N \ln 2/\lambda.$$

For example, a 1000-node network churning with median session times of one hour will see one node arrive (and one leave) every 5.2 seconds. In our experiments, we used churn rates ranging from 8/second to 4/minute, equal to median session times from 1.4 minutes to 3 hours.

Each live node continually performs lookups for identifiers chosen uniformly at random, timed by a Poisson process with rate 0.1/second, for an aggregate system load of 100 lookups/second. Each lookup is simultaneously performed by ten nodes, and we report both whether it completes and whether it is consistent with the others for the same key. If there is a majority among the ten results for a given key, all nodes in the majority are said to see a consistent result, and all others are considered inconsistent. If there is no majority, all nodes are said to see inconsistent results. This metric of consistency is more strict than that required by some DHT applications. However, both MIT's Chord and our Bamboo implementation show at least 99.9% consistency under 47-minute median session times [23], so it does not seem unreasonable.

There are two ways in which lookups fail in our tests. First, we do not perform end-to-end retries, so a lookup may fail to complete if a node in the middle of the lookup path leaves the network before forwarding the lookup request to the next node. We observed this behavior primarily in FreePastry as described below. Second, a lookup may return inconsistent results. Such failures occur either because a node is not aware of the correct node to forward the lookup to, or because it erroneously believes the correct node has left the network (because of congestion or poorly chosen timeouts). All DHT implementations we have tested show some inconsistencies under churn, but carefully chosen timeouts and judicious bandwidth usage can minimize them.

3.3 Existing DHTs

In this section we report the results of testing two mature DHT implementations under churn. Our intent here is not to place a definitive bound on the performance of either implementation. Rather, it is to motivate our work by demonstrating that handling churn in DHTs is both an important and a non-trivial problem. While we have discussed these experiments extensively with the authors of

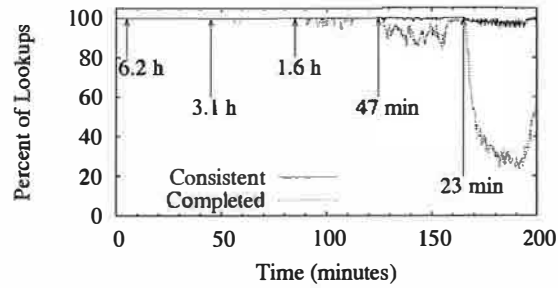


Figure 6: *FreePastry* under churn. The percentage of successful lookups in a 1000-node FreePastry network under churn. Session times for each 30-minute churn period are indicated by arrows, and each churn period is separated from the next by 10 minutes of no churn. The churn rate doubles with each successive period.

both systems, it is still possible that alternative configurations could have improved their performance. Moreover, both systems have seen subsequent development, and newer versions may show improved resilience under churn.

FreePastry We tested FreePastry 1.3, the Rice University implementation of Pastry [1]. Figure 6 shows one effect of churn on a network of 1000 FreePastry nodes, which we ran using the default 24-node leaf sets and logarithm base of 16. We do not enforce proximity between a new node and its gateway, as suggested for best FreePastry performance; this decision only effects the proximity of a node's neighbors, not the efficiency of its routing.

It is clear from Figure 6 that while successful lookups are mostly consistent, FreePastry fails to complete a majority of lookup requests under heavy churn. A likely explanation for this failure is that nodes wait so long on lookup requests to time out that they frequently leave the network with several requests still in their queues. This behavior is probably exacerbated by FreePastry's use of Java RMI over TCP as its message transport, and the way that FreePastry nodes handle the loss of their neighbors. We present evidence to support these ideas in Section 4.1.

We make a final comment on this graph. FreePastry generally recovers well between churn periods, once again correctly completing all lookups. The difficulty with real systems is that there is no such quiet period; the network is in a continual state of churn.

MIT Chord We tested MIT's Chord implementation [4] using a CVS snapshot from 8/4/2003, with the default 10-node successor lists and with the location cache disabled (using the `-F` option), since the cache causes poor performance under churn.

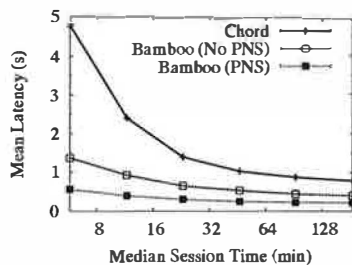


Figure 7: *Chord under churn*. Shown is the mean latency of lookups in a 1000-node MIT Chord network under increasing levels of churn. Churn increases to the left.

In contrast to FreePastry, almost all lookups in a Chord network complete and return consistent results. Chord's shortcoming under churn is in lookup latency, as shown in Figure 7, which shows the result of running Chord under the same workload as shown in Figure 6, but where we have averaged the lookup latency over each churn period. Shown for comparison are two lines representing Bamboo's performance in the same test, with and without proximity neighbor selection (PNS). Under all churn rates, Bamboo is using slightly under 750 bytes per second per node, while Chord is using slightly under 2,400.

We discuss in detail the differences that enable Bamboo to outperform Chord in Sections 4.2 and 4.3, but some of the difference in latency between Bamboo and Chord is due to their routing styles. Bamboo performs lookups recursively, whereas Chord routes iteratively. Chord could easily be changed to route recursively; in fact, newer versions of Chord support both recursive routing and PNS. Note, however, that Chord's latency grows more quickly under increasing churn than does Bamboo's. In Section 4.2, we will show evidence to support our belief that this growth is due to Chord's method of choosing timeouts for lookup messages and is independent of the lookup style employed.

3.3.1 Summary

To summarize this section, we note that we have observed several effects of churn on existing DHT implementations. A DHT may fail to complete lookup requests altogether, or it may complete them but return inconsistent results for the same lookup launched from different source nodes. On the other hand, a DHT may continue to return consistent results as churn rates increase, but it may suffer from a dramatic increase in lookup latency in the process.

4 Handling Churn

Having briefly described the way in which DHTs perform lookups, and having given evidence indicating that their ability to do so is hindered under churn, we now turn to the heart of this paper: a study of the factors contributing to this difficulty, and a comparison of solutions that can be used to overcome them. In turn, we discuss reactive versus periodic recovery from neighbor failure, the calculation of good timeout values for lookup messages, and techniques to achieve proximity in neighbor selection. The remainder of this paper focuses only on the Bamboo DHT, in which we have implemented each alternative design choice studied here. Working entirely within a single implementation allows us to minimize the differences between experiments comparing one design choice to another.

4.1 Reactive vs. Periodic Recovery

Early implementations of Bamboo suffered performance degradation under churn similar to that of FreePastry. MIT Chord's performance, however, does not degrade in the same way. A significant difference in its behavior is a design choice about how to handle detected node failures. We will call the two alternative approaches reactive and periodic recovery.

Reactive recovery In reactive recovery, a node reacts to the loss of one of its existing leaf set neighbors (or the appearance of a new node that should be added to its leaf set) by sending a copy of its new leaf set to every node in it. To save bandwidth, a node can only send differences from the last message, but the total number of messages is still $O(k^2)$ for a leaf set of k nodes. This algorithm converges quickly, is used in FreePastry, and was used in early versions of Bamboo. MSPastry uses a more bandwidth-efficient, but more complex, variant of reactive recovery [7].

Periodic recovery In contrast, in periodic recovery a node periodically shares its leaf set with each of the members of that set, each of whom responds in kind with its own leaf set. The process takes place independently of the node detecting changes in its leaf set. As a simple optimization, a node picks one random member of its leaf set to share state with in each period. This change saves bandwidth, but still converges in $O(\log k)$ phases, where k is the size of the leaf set. (Further details can be found elsewhere [23].) This algorithm is the one currently used by Bamboo, and the periodic nature of this algorithm is shared by Chord's method of keeping its successor list correct.

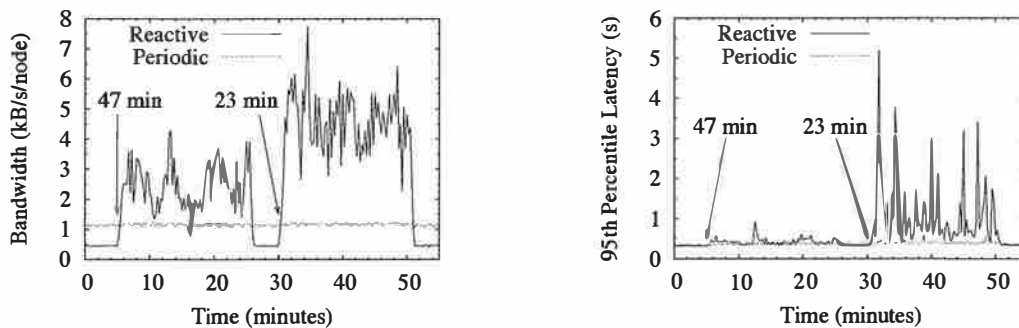


Figure 8: *Reactive versus periodic recovery.* Without churn, reactive recovery is very efficient, as messages are only sent in response to actual changes. At reasonable churn rates, however, periodic recovery uses less bandwidth, and lower contention for the network leads to lower latencies.

4.1.1 Positive feedback cycles

Reactive recovery runs the risk of creating a positive feedback cycle as follows. Consider a node whose access link to the network is sufficiently congested that timeouts cause it to believe that one of its neighbors has failed. If the node is recovering reactively, recovery operations begin, and the node will add even more packets to its already congested network link. This added congestion will increase the likelihood that the node will mistakenly conclude that other neighbors have failed. If this process continues, the node will eventually cause congestion collapse on its access link.

Observations of these cycles in early Bamboo (and examination of the Chord code) originally led us to propose periodic recovery for handling churn. By decoupling the rate of recovery from the discovery of failures, periodic recovery prevents the feedback cycle described above. Moreover, by lengthening the recovery period with the observation of message timeouts, we can introduce a negative feedback cycle, further improving resilience.

Another way to mitigate the instability associated with reactive recovery is to be more conservative when detecting node failure. We have found one effective approach to be to conclude failure only after 15 consecutive message timeouts to a neighbor. Since timeouts are backed off multiplicatively to a maximum of five seconds, it is unlikely that a node will conclude failure due to congestion. One drawback with this technique, however, is that neighbors that have actually failed remain in a node's routing table for some time. Lookups that would route through these neighbors are thus delayed, resulting in long lookup latencies. To remedy this problem, a node stops routing through a neighbor after seeing five consecutive message timeouts to that neighbor. We have found these changes make reactive recovery feasible for small leaf sets and moderate churn.

4.1.2 Scalability

Experiments show little difference in correctness between periodic and reactive recovery. To see why, consider a node *A* that joins a network, and let *B* be the node in the existing network whose identifier most closely matches that of *A*. As in Pastry, *A* retrieves its initial leaf set by contacting *B*, and *B* adds *A* to its leaf set immediately after confirming its IP address and port (with a probe message). Until *A*'s arrival propagates through the network, another node *C* may still route messages that should go to *A* to *B* instead, but *B* will just forward these messages on to *A*. Likewise, should *A* fail, *B* will still be in *C*'s leaf set, so once routing messages to *A* time out, *C* and other nearby nodes will generally all agree that *B* is the next best choice.

While both periodic and reactive recovery achieve roughly identical correctness, there is a large difference in the bandwidth consumed under different churn rates and leaf set sizes. (A commonly accepted rule of thumb is that to provide sufficient resilience to massive node failure, the size of a node's leaf set should be logarithmic in the system size.) Under low churn, reactive recovery is very efficient, as messages are only sent in response to actual changes, whereas periodic recovery is wasteful. As churn increases, however, reactive recovery becomes more expensive, and this behavior is exacerbated by increasing leaf set size. Not only does a node see more failures when its leaf set is larger, but the set of other nodes it must notify about the resulting changes in its own leaf set is larger. In contrast, periodic recovery aggregates all changes in each period into a single message.

Figure 8 shows this contrast in Bamboo using leaf sets of 24 nodes, the default leaf set size in FreePastry. In this figure, we ran Bamboo using both configurations for two 20-minute churn periods of 47 and 23 minute median session times separated by five minutes with no churn.

We note that during the periods of the test where there

is no churn, reactive recovery uses less than half of the bandwidth of periodic recovery. On the other hand, under churn its bandwidth use jumps dramatically. As discussed above, Bamboo does not suffer from positive feedback cycles on account of this increased bandwidth usage. Nevertheless, the extra messages sent by reactive recovery compete with lookup messages for the available bandwidth, and as churn increases we see a corresponding increase in lookup latency. Although not shown in the figure, the number of hops per lookup is virtually identical between the two schemes, implying that the growth in bandwidth is most likely due to contention for the available bandwidth.

Since our goal is to handle median session times down to a few minutes with low lookup latency, we do not explore reactive recovery further in this work. The remainder of the Bamboo results we present are all obtained using periodic recovery.

4.2 Timeout Calculation

In this section, we discuss the role that timeout calculation on lookup messages plays in handling churn.

To understand the relative importance of timeouts in a DHT as opposed to a more traditional networked system, consider a traditional client-server system such as the networked file system (NFS). In NFS, the server does not often fail, and when it does there are generally few options for recovery and no alternative servers to fail over to. If a response to an NFS request is not received in the expected time, the client will usually try again with an exponentially increasing timeout value.

In a peer-to-peer system under churn, in contrast, requests will be frequently sent to a node that has left the system, possibly forever. At the same time, a DHT with routing flexibility (static resilience) has many alternate paths available to complete a lookup. Simply backing off the request period is thus a poor response to a request timeout; it is often better to retry the request through a different neighbor.

A node should ensure that the timeout for a request was judiciously selected before routing to an alternate neighbor. If it is too short, the node to which the original was sent may be yet to receive it, may be still processing it, or the response may be queued in the network. If so, injecting additional requests may result in the use of additional bandwidth without any beneficial result—for example, in the case that the local node's access link is congested. Conversely, if the timeout is too long, the requesting node may waste time waiting for a response from a node that has left the network. If the request rate is fixed at too low a value, these long waits cause unbounded queue growth on the request node that might be avoided with shorter timeouts.

For these reasons, nodes should accurately choose

timeouts such that a late response is indicative of node failure, rather than network congestion or processor load.

4.2.1 Techniques

We discuss and study three alternative timeout calculation strategies. In the first, we fix all timeouts at a conservative value of five seconds as a control experiment. In the second, we calculate TCP-style timeouts using direct measurement of past response times. Finally, we explore using indirect measurements from a virtual coordinate algorithm to calculate timeouts.

TCP-style timeouts: If a DHT routes recursively, it rarely communicates with nodes other than its direct neighbors in the overlay network. Since the number of these neighbors is logarithmic in the size of the network, and since each node periodically probes each neighbor for availability, a node can easily maintain a past history of each neighbor's response times for use in calculating timeouts. In Bamboo, we have implemented this strategy following the style of the early TCP work [15], where each node maintains an exponentially weighted mean and variance of the response time for each neighbor. Specifically, the estimate round-trip timeout (RTO) for a neighbor is calculated as

$$\text{RTO} = \text{AVG} + 4 \times \text{VAR},$$

where AVG is the observed average round-trip time and VAR is the observed mean variance of that time.

Timeouts from virtual coordinates: In contrast to recursive routing, with iterative routing a node must potentially have a good timeout for *any* other node in the network. However, in some scenarios iterative routing does have attractive properties. For example, since the source of a lookup request controls the entire process of iterative routing, it is easy to explore several different lookup paths in parallel. For only a constant increase in bandwidth used, this technique prevents a single timeout from delaying a lookup [16].

Virtual coordinates provide one approach to computing timeouts without previously measuring the response time to every node in the system. In this scheme, a distributed machine learning algorithm is employed to assign to each node coordinates in a virtual metric space such that the distance between two nodes in the space is proportional to their latency in the underlying network.

Bamboo includes an implementation of the Vivaldi coordinate system employed by Chord [11]. Vivaldi keeps an exponentially-weighted average of the error of past round-trip times calculated with the coordinates, and computes the RTO as

$$\text{RTO} = v + 6 \times \alpha + 15$$

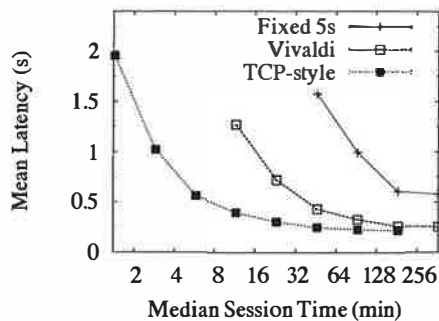


Figure 9: TCP-style versus virtual coordinate-based timeouts in Bamboo. Timeouts chosen using Vivaldi are competitive with TCP-style timeouts for moderate churn rates.

where v is the predicted round-trip time and α is the average error. The constant term of 15 milliseconds is added to avoid unnecessary retransmissions when the destination is the local host.

4.2.2 Results

TCP-style timeouts assume a recursive routing algorithm, and a virtual coordinate system is necessary only when routing iteratively. While we would ideally compare the two approaches by measuring each in its intended environment, this would prevent us from isolating the effect of timeouts from the differences caused by routing styles.

Instead, we study both schemes under recursive routing. If timeouts calculated with virtual coordinates provide performance comparable to those calculated in the TCP-style under recursive routing, we can expect the virtual coordinate scheme to not be prohibitively expensive under iterative routing. While other issues may remain with iterative routing under churn (e.g. congestion control—see Section 6), this result would be a useful one.

Figure 9 shows a direct comparison of the three timeout calculation methods under increasing levels of churn. In all cases in this experiment, the Bamboo configurations differed only in choice of timeout calculation method. Proximity neighbor selection was used, but the latency measurements for PNS used separate direct probing and not the virtual coordinates.

Even under light levels of churn, fixing all timeouts at five seconds causes lookup timeouts to pull the mean latency up to more than twice that of the other configurations, confirming our intuition about the importance of good timeout values in DHT routing under churn. Moreover, by comparing Figure 9 to Figure 7, we note that under high churn timeout calculation has a greater effect on lookup latency than the use of PNS.

Virtual coordinate-based timeouts achieve very similar mean latency to TCP-style timeouts at low churn. Fur-

thermore, they perform within a factor of two of TCP-style measurements until the median churn rate drops to 23 minutes. Past this point, their performance quickly diverges, but virtual coordinates continue to provide mean lookup latencies under two seconds down to twelve-minute median session times.

Finally, we note the similarity in shape of Figure 9 to Figure 7, where we compared the performance of Chord to Bamboo, suggesting that the growth in lookup latency under Chord at high churn rates is due to timeout calculation based on virtual coordinates.

4.3 Proximity Neighbor Selection

Perhaps one of the most studied aspects of DHT design has been proximity neighbor selection (PNS), the process of choosing among the potential neighbors for any given routing table entry according to their network latency to the choosing node. This research is well motivated. The *stretch* of a lookup operation is defined as the latency of the lookup divided by the round-trip time between the lookup source and the node discovered by the lookup in the underlying IP network. Dabek et al. present an argument and experimental data that suggest that PNS allows a DHT of N nodes to achieve median stretch of only 1.5, independent of the size of the network and despite using $O(\log N)$ hops [11]. Others have proved that PNS can be used to provide constant stretch in locating replicas under a restricted network model [21]. This is the first study of which we are aware, however, to compare methods of achieving PNS under churn. We first take a moment to discuss the common philosophy and techniques shared by each of the algorithms we study.

4.3.1 Commonalities

One of the earliest insights in DHT design was the separation of correctness and performance in the distinction between neighbors in the leaf set and neighbors in the routing table [24, 27]. So long as the leaf sets in the system are correct, lookups will always return correct results, although they may take $O(N)$ hops to do so. Leaf set maintenance is thus given priority over routing table maintenance by most DHTs. In the same manner, we note that so long as each entry in the routing table has *some* appropriate neighbor (i.e. one with the correct identifier prefix), lookups will always complete in $O(\log N)$ hops, even though they may take longer than if the neighbors had been chosen for proximity. We say such lookups are *efficient*, even though they may not have low stretch. By this argument, we reason that it is desirable to fill a routing table entry quickly, even with a less than optimal neighbor; finding a nearby neighbor is a lower priority.

There is a further argument to treating proximity as a

lower priority in the presence of churn. Since we expect our set of neighbors to change over time as part of the churn process, it makes little sense to work too hard to find the absolute closest neighbor at any given time; we might expend considerable bandwidth to find them only to see them leave the network shortly afterward. As such, our general approach is to run each of the algorithms described below *periodically*. In the case where churn is high, this technique allows us to retune the routing table as the network changes. When churn is low, rerunning the algorithms makes up for latency measurement errors caused by transient network conditions in previous runs.

Our general approach to finding nearby neighbors thus takes the following form. First, we use one of the algorithms below to find nodes that may be near to the local node. Next, we measure the latency to those nodes. If we have no existing neighbor in the routing table entry that the measured node would fill, or if it is closer than the existing routing table entry, we replace that entry, otherwise we leave the routing table unchanged. Although the bandwidth cost of multiple measurements is high, the storage cost to remember past measurements is low. As a compromise, we perform only a single latency measurement to each discovered node during any particular run of an algorithm, but we keep an exponentially weighted average of past measurements for each node, and we use this average value in deciding the relative closeness of nodes. This average occupies only eight bytes of memory for each measured node, so we expect this approach to scale comfortably to very large systems.

4.3.2 Techniques

The techniques for proximity neighbor selection that we study here are global sampling, sampling of our neighbors' neighbors, and sampling of the nodes that have our neighbors as their neighbors. We describe each of these techniques in turn.

Global sampling In global sampling (called *global tuning* in our earlier work [23]), we use the lookup functionality of the DHT to find new neighbors. For a routing table entry that requires a neighbor with prefix p , we perform a lookup for a random identifier with prefix p . The node returned by this lookup will almost always have the desired prefix. (As an example of why this is not always the case, note that a lookup of identifier 0 may return a node whose identifier starts with 1 if the node with the largest identifier in the ring is numerically closer to 0 than the node with the smallest identifier.) Given enough samples, all nodes with prefix p will eventually be probed. The motivation for this technique comes from Gummadi et al., who showed that sampling only around 16 nodes for each routing table entry provides almost optimal proximity [12].

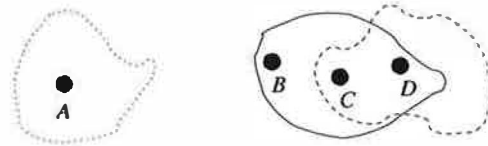


Figure 10: *Sampling neighbors' neighbors*. If A joins using D as its gateway, its initial level-0 neighbors are the same as those of D ; assume that these are all within the dashed line. A contacts a level-0 neighbor, e.g. C , and asks it for its level-0 neighbors. A would learn about B in this manner. However, there may be no path from the D 's ideal neighbors to those of A .

There are some cases, however, where global sampling will take unreasonably long to find the closest possible neighbor. For example, consider two nodes separated from the core Internet by the same, high latency access link, as shown in Figure 11. The relatively high latency seen by these two nodes to all other nodes in the network makes them attractive neighbors for each other; if they have different first digits in a network with logarithm base two, they can drastically reduce the cost of the first hop of many routes by learning about each other. However, the time for these nodes to find each other using global sampling is proportional to the size of the total network, and so they may not find each other before their sessions end. It is this drawback of global sampling that leads us to consider other techniques.

Neighbors' neighbors The next technique we consider is sampling our neighbors neighbors, a process called *routing table maintenance* in the Pastry work [24] or *local tuning* in our earlier work [23]. In this technique, we contact an existing routing table neighbor at level l of our routing table and ask for its level l neighbors. Like us, these nodes share a prefix of $l - 1$ digits with the contacted neighbor and are thus appropriate for use in our routing table as well. As in global sampling, having discovered these new nodes, we probe them for latency and use them if they are closer than our existing neighbors.

The motivation for sampling neighbors' neighbors is illustrated in Figure 10; it relies on the expectation that proximity in the network is roughly transitive. If a node discovers one nearby node, then that node's neighbors are probably also nearby. In this way, we expect that a node can "walk" through the graph of neighbor links to the set of nodes most near it.

To see one possible shortcoming of sampling our neighbors' neighbors, consider again Figure 11. While the two isolated nodes would like to discover each other, it is unlikely that any other nodes in the network would prefer them as neighbors; their isolation makes them unattractive for routing lookups that originate elsewhere, except

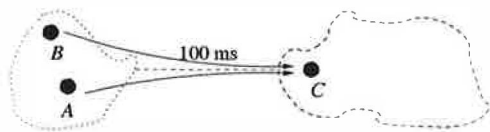


Figure 11: *Sampling neighbors' inverse neighbors.* Nodes A and B are isolated from the remainder of the network by a long latency, and are initially unaware of each other. Such a situation is possible if, for example, two European nodes join a network of primarily North American nodes. As such, they make unattractive neighbors for other nodes, but they would still like to find each other. If they both have C as a neighbor, they can find each other by asking C for its inverse neighbors.

in the rare case that they are the result of those lookups. As such, since neighbor links in DHTs are rarely symmetric, it is unlikely that there is a path through the graph of neighbor links that will lead one isolated node to the other, despite their relative proximity.

Neighbors' inverse neighbors The latter argument presents an obvious alternative approach. Instead of sampling our neighbors' neighbors, why not sample those nodes which have the same neighbors as the local node? This technique was originally proposed in the Tapestry nearest neighbor algorithm [14]; we call it sampling our neighbors' inverse neighbors. To motivate this technique, consider again Figure 11. Although the two remote nodes are unlikely to be neighbors of many other nodes, given that their existing neighbors are mostly nearby, they are likely to choose the same neighbors from outside their isolated domain. For this reason, they are likely to find each other in the set of their neighbors' inverse neighbors.

Normally, a DHT node would not record the set of nodes that use it as a neighbor. Actively managing such a list, in fact, requires additional probing bandwidth. Currently, the Bamboo implementation does actively manage this set, but it could be easily approximated at each node by keeping track of the set of nodes which have sent it liveness probes in the last minute or so. We plan to implement this optimization in our future work.

Recursive sampling Consider Figure 11 one final time, and assume that we are using a single-bit digits and that the two remote nodes begin with different digits, i.e. 0 and 1 respectively. The node whose identifier starts with 0 will have only one neighbor whose identifier begins with 1 (its level-0 neighbor). Likewise, the node whose identifier starts with 1 will have only one neighbor that starts with 0. The set of neighbors in whose inverse neighbor sets the two isolated neighbors can find each other is thus very small. As such, until the two isolated nodes have found

```
(1) function nearest_neighbors () =
(2)   S = highest_nonempty_rt_level ();
(3)   l = longest_matching_prefix (S);
(4)   while l >= 0
(5)     forall n in S
(6)       T = n.get_inverse_rt_neighbors (l);
(7)       S = closest (k, S ∪ T);
```

Figure 12: *The Tapestry nearest neighbor algorithm.*

very nearby level-0 neighbors, they will be unlikely to find each other among their neighbors' inverse neighbors.

To remedy this final problem, we can perform the sampling of nodes in a manner similar to that used by the Tapestry nearest neighbor algorithm (and the Pastry optimized join algorithm). Pseudo-code for this technique is shown in Figure 12. Starting with the highest level l in its routing table, a node contacts the neighbors at that level and retrieves their neighbors (or inverse neighbors). The latency to each newly discovered nodes is measured, and all but the k closest are discarded. The node then decrements l and retrieves the level- l neighbors from each non-discarded node. This process is repeated until $l < 0$. Along the way, each discovered neighbor is considered as a candidate for use in the routing table. To keep the cost of this algorithm low, we limit it to having at most three outstanding messages (neighbor requests or latency probes) at any time.

Note that although this process starts by sampling from the routing table, the set of nodes on which it recurses is not constrained by the prefix-matching structure of that table. As such, it does not suffer from the small rendezvous set problem discussed above. In fact, under certain network assumptions, it has been proved that this process finds a node's nearest neighbor in the underlying network.

4.3.3 Results

In order to compare the techniques described above, it is important to consider not only effective they are at finding nearby neighbors, but also at what bandwidth cost they do so. For example, global sampling at a high enough rate relative to the churn rate would achieve perfect proximity, but at the cost of a very large number of lookups and latency probes. To make this comparison, then, we ran each algorithm (and some combinations of them) at various periods, then plotted the mean lookup latency under churn versus bandwidth used. The results for median session times of 47 minutes are shown in Figure 13, which is split into two graphs for clarity.

Figure 13(a) shows several interesting results. First, we note that only a little bit of global sampling is necessary to produce a drastic improvement in latency versus the

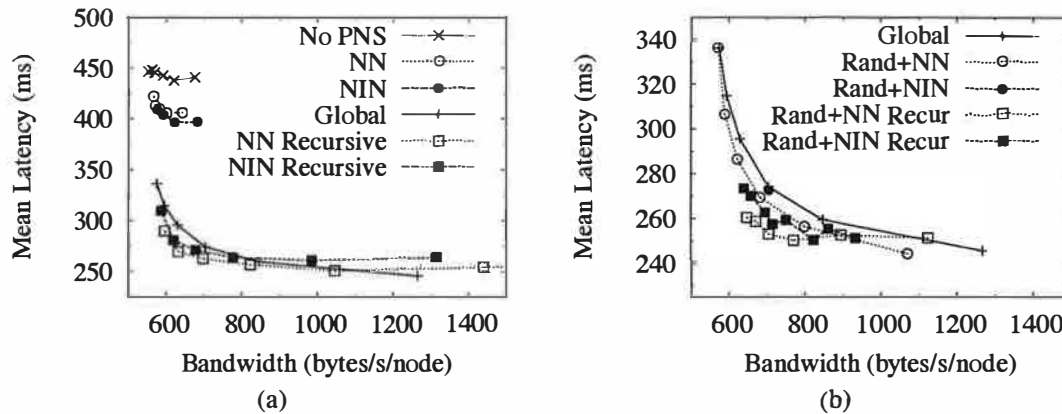


Figure 13: *Comparison of PNS techniques.* “No PNS” is the control case, where proximity is ignored. “Global Sampling” uses the lookup function to sample all nodes in the DHT. “NN” is sampling our neighbor’s neighbors, and “NIN” is sampling their inverse neighbors. The recursive versions of “NN” and “NIN” mimic the nearest-neighbor algorithms of Pastry and Tapestry, respectively. Note that the scales are different between the two figures.

configuration that is not using PNS. With virtually no increase in bandwidth, global sampling drops the mean latency from 450 ms to 340 ms.

Next, much to our surprise, we find that simple sampling of our neighbor’s neighbors or inverse neighbors is not terribly effective. As we argued above, this result may be in part due to the constraints of the routing table, but we did not expect the effect to be so dramatic. On the other hand, the recursive versions of both algorithms are at least as effective as global sampling, but not much more so. This result agrees with the contention of Gummadi et al. that only a small amount of global sampling is necessary to achieve near-optimal PNS.

Figure 13(b) shows several combinations of the various algorithms. Global sampling plus sampling of neighbors’ neighbors—the combination used in our earlier work [23]—does well, offering a small decrease in latency without much additional bandwidth. However, the other combinations offer similar results. At this point, it seems prudent to say that the most effective technique is to combine global sampling with any other technique. While there may be other differences between the techniques not revealed by this analysis, we see no clear reason to prefer one over another as yet.

5 Related Work

As we noted at the start of this paper, while DHTs have been the subject of much research in the last 4 years or so, there have been few studies of the resilience of real implementations at scale, perhaps because of the difficulty of deploying, instrumenting, and creating workloads for such deployments. However, there has been a substantial amount of theoretical and simulation-based work.

Gummadi et al. [12] present a comprehensive analysis of the static resilience of the various DHT geometries. As we have argued earlier in this work, static resilience is an important first step in a DHT’s ability to handle failures in general and churn in particular.

Liben-Nowell et al. [17] present a theoretical analysis of structured peer-to-peer overlays from the point of view of churn as a continuous process. They prove a lower bound on the maintenance traffic needed to keep such networks consistent under churn, and show that Chord’s algorithms are within a logarithmic factor of this bound. This paper, in contrast, has focused more on the systems issues that arise in handling churn in a DHT. For example, we have observed what they call “false suspicions of failure”, the appearance that a functioning node has failed, and shown how reactive failure recovery can exacerbate such conditions.

Mahajan et al. [19] present a simulation-based analysis of Pastry in which they study the probability that a DHT node will forward a lookup message to a failed node as a function of the rate of maintenance traffic. They also present an algorithm for automatically tuning the maintenance rate for a given failure rate. Since this algorithm increases the rate of maintenance traffic in response to losses, we are concerned that it may cause positive feedback cycles like those we have observed in reactive recovery. Moreover, we believe their failure model is pessimistic, as they do not consider hop-by-hop retransmissions of lookup messages. By acknowledging lookup messages on each hop, a DHT can route around failed nodes in the middle of a lookup path, and in this work we have shown that good timeout values can be computed to minimize the cost of such retransmissions.

Castro et al. [7] presented a number of optimizations

they have performed in MSPastry, the Microsoft Research implementation of Pastry, using simulations. Also, Li et al. [16] performed a detailed simulation-based analysis of several different DHTs under churn, varying their parameters to explore the latency-bandwidth tradeoffs presented. It was their work that inspired our analysis of different PNS techniques.

As opposed to the emulated network used in this study, simulations do not usually consider such network issues as queuing, packet loss, etc. By not doing so, they either simulate far larger networks than we have studied here as in [7, 19], or they are able to explore a far larger space of possible DHT configurations as in [16]. On the other hand, they do not reveal subtle issues in DHT design, such as the tradeoffs between reactive and periodic recovery. Also, they do not reveal the interactions of lookup traffic and maintenance traffic in competing for network bandwidth. We are interested in whether a useful middle ground exists between these approaches.

Finally, a number of useful features for handling churn have been proposed, but are not implemented by Bamboo. For example, Kademlia [20] maintains several neighbors for each routing table entry, ordered by the length of time they have been neighbors. Newer nodes replace existing neighbors only after failure of the latter. This design decision is aimed at mitigating the effects of the high “infant mortality” observed in peer-to-peer networks.

Another approach to handling churn is to introduce a hierarchy into the system, through stable “superpeers” [2, 29]. While an explicit hierarchy is a viable strategy for handling load in some cases, this work has shown that a fully decentralized, non-hierarchical DHT can in fact handle high rates of churn at the routing layer.

6 Future Work

As discussed in the introduction, there are several other limitations of this study that we think provide for important future work. At an algorithmic level, we would like to study the effects of alternate routing table neighbors as in Kademlia and Tapestry. We would also like to continue our study of iterative versus recursive routing. As discussed by others [11], congestion control for iterative lookups is a challenging problem. We have implemented Chord’s STP congestion control algorithm and are currently investigating its behavior under churn, but we do not yet have definitive results about its performance.

At a methodological level, we would like to broaden our study to include better models of network topology and churn. We have so far used only a single network topology in our work, and so our results should be not be taken as the last word on PNS. In particular, the distribution of internode latencies in our ModelNet topology

is more Gaussian than the distribution of latencies measured on the Internet. Unfortunately for our purposes, these measured latency distributions do not include topology information, and thus cannot be used to simulate the kind of network cross traffic that we have found important in this study. The existence of better topologies would be most welcome.

In addition to more realistic network models, we would also like to include more realistic models of churn in our future work. One idea that was suggested to us by an anonymous reviewer was to scale traces of session times collected from deployed networks to produce a range of churn rates with a more realistic distribution. We would like to explore this approach. Nevertheless, we believe that the effects of the factors we have studied are dramatic enough that they will remain important even as our models improve.

Finally, in this work we have only shown the resistance of the Bamboo *routing* layer to churn, an important first step verifying that DHTs are ready to become the dominant building block for peer-to-peer systems, but a limited one. Clearly other issues remain. Security and possibly anonymity are two such issues, but we are unclear about how they relate to churn. We are currently studying the resilience to churn of the algorithms used by the DHT *storage* layer. We hope that the existence of a routing layer that is robust under churn will provide a useful substrate on which these remaining issues may be studied.

7 Conclusion

In this work we have summarized the rates of churn observed in deployed peer-to-peer systems and shown that existing DHTs exhibit less than desirable performance at the higher end of these churn rates. We have presented Bamboo and explored various design tradeoffs and their effects on its ability to handle churn.

The design tradeoffs we studied in this work fall into three broad categories: reactive versus periodic recovery from neighbor failure, the calculation of timeouts on lookup messages, and proximity neighbor selection. We have presented the danger of positive feedback cycles in reactive recovery and discussed two ways to break such cycles. First, we can make the DHT much more cautious about declaring neighbors failed, in order to limit the possibility that we will be tricked into recovering a non-faulty node by network congestion. Second, we presented the technique of periodic recovery. Finally, we demonstrated that reactive recovery is less efficient than periodic recovery under reasonable churn rates when leaf sets are large, as they would be in a large system.

With respect to timeout calculation, we have shown that TCP-style timeout calculation performs best, but argued

that it is only appropriate for lookups performed recursively. It has long been known that recursive routing provides lower latency lookups than iterative, but this result presents a further argument for recursive routing where the lowest latency is important. However, we have also shown that while they are not as effective as TCP-style timeouts, timeouts based on virtual coordinates are quite reasonable under moderate rates of churn. This result indicates that at least with respect to timeouts, iterative routing should not be infeasible under moderate churn.

Concerning proximity neighbor selection, we have shown that global sampling can provide a 24% reduction in latency for virtually no increase in bandwidth used. By using an additional 40% more bandwidth, a 42% decrease in latency can be achieved. Other techniques are also effective, especially our adaptations of the Pastry and Tapestry nearest-neighbor algorithms, but not much more so than simple global sampling. Merely sampling our neighbors' neighbors or inverse neighbors is not very effective in comparison. Some combination of global sampling and any of the other techniques seems to provide the best performance at the least cost.

8 Acknowledgments

We would like to thank a number of people for their help with this work. Our shepherd, Atul Adya, and the anonymous reviewers all provided valuable comments and guidance. Frank Dabek helped us tune our Vivaldi implementation, and he and Emil Sit helped us get Chord up and running. Likewise, Peter Druschel provided valuable debugging insight for FreePastry. David Becker helped us with ModelNet. Sylvia Ratnasamy, Scott Shenker, and Ion Stoica provided valuable guidance at several stages of this paper's development.

References

- [1] Freepastry 1.3. <http://www.cs.rice.edu/CS/Systems/Pastry/>.
- [2] Gnutella. <http://www.gnutella.com/>.
- [3] Inet topology generator. <http://topology.eecs.umich.edu/inet/>.
- [4] MIT Chord. <http://www.pdos.lcs.mit.edu/chord/>.
- [5] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. IPTPS*, Feb. 2003.
- [6] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. 2003.
- [7] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft, 2003.
- [8] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. Apr. 2003.
- [9] J. Chu, K. Labonte, and B. N. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proc. of ITCom: Scalability and Traffic Control in IP Networks*, July 2002.
- [10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP*, Oct. 2001.
- [11] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. NSDI*, 2004.
- [12] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM*, Aug. 2003.
- [13] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. ACM SOSP*, Oct. 2003.
- [14] K. Hildrum, J. D. Kubiawicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. SPAA*, 2002.
- [15] V. Jacobson and M. J. Karels. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, 1988.
- [16] J. Li, J. Stribling, T. M. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. IPTPS*, 2004.
- [17] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proc. ACM PODC*, July 2002.
- [18] B. T. Loo, R. Huebsch, I. Stoica, and J. Hellerstein. The case for a hybrid P2P search infrastructure. In *Proc. IPTPS*, 2004.
- [19] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proc. IPTPS*, Feb. 2003.
- [20] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS*, 2002.
- [21] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM SPAA*, June 1997.
- [22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, Aug. 2001.
- [23] S. Rhea, D. Geels, T. Roscoe, and J. Kubiawicz. Handling churn in a DHT. Technical Report UCB/CSD-03-1299, University of California, Berkeley, December 2003.
- [24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, Nov. 2001.
- [25] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN*, Jan. 2002.
- [26] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proc. of ACM SIGCOMM Internet Measurement Workshop*, Nov. 2002.
- [27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, Aug. 2001.
- [28] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI*, Dec. 2002.
- [29] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiawicz. Brocade: Landmark routing on overlay networks. In *Proc. IPTPS*, March 2002.
- [30] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiawicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 22(1):41–53, Jan. 2004.

A Network Positioning System for the Internet *

T. S. Eugene Ng

*Department of Computer Science
Rice University*

Hui Zhang

*Department of Computer Science
Carnegie Mellon University*

Abstract

Network positioning has recently been demonstrated to be a viable concept to represent the network distance relationships among Internet end hosts. Several subsequent studies have examined the potential benefits of using network position in applications, and proposed alternative network positioning algorithms. In this paper, we study the problem of designing and building a network positioning system (NPS). We identify several key system-building issues such as the consistency, adaptivity and stability of host network positions over time. We propose a hierarchical network positioning architecture that maintains consistency while enabling decentralization, a set of adaptive decentralized algorithms to compute and maintain accurate, stable network positions, and finally present a prototype system deployed on PlanetLab nodes that can be used by a variety of applications. We believe our system is a viable first step to provide a network positioning capability in the Internet.

1 Introduction

Network distance, i.e. round-trip propagation and transmission delay¹, is a fundamental property of a network path that affects application performance. Our initial Internet measurement study [15] and subsequent measurement-based studies [22][6] have shown that by knowing the network distances from end hosts to a few, say 20, other hosts, it is feasible to characterize the end hosts' *network positions* as points in a low-dimensional Euclidean space model (say 6 dimensions) such that the Euclidean distance between two end hosts' network positions accurately predicts the actual Internet network distance in most cases. In short, network position is a feasible concept and can represent Internet network distance relationships efficiently.

*This research was sponsored by DARPA under contract number F30602-99-1-0518, by NSF under grant numbers Career Award NCR-9624979, ANI-9730105, ITR Award ANI-0085920, and ANI-9814929, and by the Texas Advanced Research Program under grant No.003604-0078-2003. Additional support was provided by Intel. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, Texas ARP, Intel, or the U.S. government.

¹Note that network distance is a property of the network topology and routing, and is not dependent on the instantaneous network load, thus it is easier to model and predict.

Network position can be very useful for a variety of wide-area network applications. For example, a CAN overlay network [18] can use network positions as logical overlay positions and reduce the average overlay delay compared to an un-optimized structure. A wide-area service location agent can also use network positions of clients and servers to direct clients to nearby servers without measuring the paths between the clients and the servers. For bandwidth demanding applications, network positions can be a highly scalable mechanism to select a small subset of nearby nodes and then bandwidth probing techniques (e.g. [12]) can be applied to only this subset to achieve higher efficiency. Several recent studies have examined the benefits of using network position in wide-area network applications [3][9][24][5]. Alternative network position computation algorithms have also been proposed [13][16][20][22][6][5].

In this paper, we study the problem of building a network positioning system (NPS) to provide a positioning capability in the Internet. To the best of our knowledge, this is the first study to consider system-building issues in network positioning. There are two alternatives to deploy an NPS, the first is to integrate an NPS with a particular application, the second is to deploy an NPS that is shared by many applications. In this paper, we consider the latter alternative. This is similar to the deployment model of the Domain Name System (DNS) which provides a naming capability that is shared by applications in the Internet. Like many other systems work, this paper addresses a broad set of issues including identifying key design issues, system architecture design, algorithms design, implementation details, system tuning, and to a limited extent security. We make the following contributions in this paper:

- Identify key system-building issues: (1) consistency of network positions, (2) adapt positions to network topology changes, (3) maintain position stability when network topology is not changing.
- A hierarchical network positioning architecture that maintains position consistency while enabling decentralization.
- A set of adaptive decentralized algorithms to compute and maintain accurate, stable network posi-

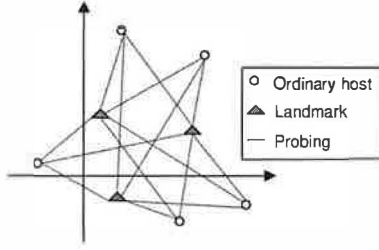


Figure 1: Basic network positioning method.

tions, even when a minority of malicious nodes may lie about their positions.

- A functional prototype deployed on PlanetLab [17] nodes that can be used by a variety of applications.

We believe our system may serve as a first step towards providing a network positioning capability in the Internet and can benefit existing and emerging applications. Our system is in a relatively early stage of development. Operational experience and application experience in the future will help us better select the system parameters and guide the evolution of the design of the system.

In the next section, we will describe the network positioning concept as originally proposed in [15] and outline the basic method for simulating network positioning in an idealized setting. We will also present some application examples. In Section 3, we present the detailed design of the system, including the architecture and algorithms. In Section 4, we describe the implementation of the system. Results from our experience with the system on PlanetLab and controlled experiments performed in a simulator are presented Section 5. We present the related work in Section 6 and finally conclude in Section 7.

2 Network Positioning

We begin the discussion by describing the concept of network positioning and the network positioning method proposed in [15]. The accuracy of the network positions computed by an idealized simulation of this method will serve as the target for the NPS to approximate.

Conceptually, network positioning seeks to embed the Internet network distance relationship into a geometric space (e.g. an Euclidean space). That is, each Internet host is assigned a position (a set of coordinates) in a geometric space model such that the Internet network distances among the hosts are as well approximated by the geometric distances in the model as possible.

The network distance between two hosts \mathcal{H}_1 and \mathcal{H}_2 , denoted by $d_{\mathcal{H}_1\mathcal{H}_2}$, is defined as the round-trip propagation delay and transmission delay of the network path between them. Operationally, it is the minimum observable round-trip time (RTT).

A method to compute an embedding is outlined in [15]. This method works as follows. N special hosts called Landmarks, denoted by $\mathcal{L}_1, \dots, \mathcal{L}_N$, are deployed in the Internet and the inter-Landmark distances are measured. The inter-Landmark distances are transmitted to a central node. The central node then computes a set of Landmark coordinates and return the coordinates to the Landmarks. The Landmark coordinates, denoted by $c_{\mathcal{L}_1}, \dots, c_{\mathcal{L}_N}$, are the result of minimizing the following objective function:

$$f_{obj1}(c_{\mathcal{L}_1}, \dots, c_{\mathcal{L}_N}) = \sum_{i,j \in \{1, \dots, N\} \mid i > j} \mathcal{E}(d_{\mathcal{L}_i\mathcal{L}_j}, \hat{d}_{\mathcal{L}_i\mathcal{L}_j}) \quad (1)$$

where $\hat{d}_{\mathcal{L}_i\mathcal{L}_j}$ is the geometric distance between $c_{\mathcal{L}_i}$ and $c_{\mathcal{L}_j}$ and $\mathcal{E}(\cdot)$ is the error measurement function:

$$\mathcal{E}(d_{\mathcal{H}_1\mathcal{H}_2}, \hat{d}_{\mathcal{H}_1\mathcal{H}_2}) = \left(\frac{d_{\mathcal{H}_1\mathcal{H}_2} - \hat{d}_{\mathcal{H}_1\mathcal{H}_2}}{d_{\mathcal{H}_1\mathcal{H}_2}} \right)^2 \quad (2)$$

The simplex downhill algorithm [14] is used to solve the optimization problem. The Landmarks' coordinates define the bases for the geometric space. To embed an ordinary host \mathcal{H} , \mathcal{H} uses the Landmarks as *reference points* and probes all the Landmarks to obtain the Landmarks' coordinates and the network distances to the Landmarks (see Figure 1). It then computes its coordinates, $c_{\mathcal{H}}$, that minimize the following objective function:

$$f_{obj2}(c_{\mathcal{H}}) = \sum_{i \in \{1, \dots, N\}} \mathcal{E}(d_{\mathcal{H}\mathcal{L}_i}, \hat{d}_{\mathcal{H}\mathcal{L}_i}) \quad (3)$$

An idealized simulation of this network positioning method is a centralized computation procedure that takes static network distance data as input and solves the optimization problems to determine a set of network positions. Using this method with Internet measurements, it has been shown in [15] that the Internet network distance relationship is accurately embeddable in a low-dimensional Euclidean space. In particular, in a 7-dimensional Euclidean space, 50% of the distance predictions have less than 10% error, 90% of them have less than 50% error, and the network positions can be used to accurately select nearby hosts in the Internet.

Example Applications Network position can be very useful in wide-area network performance optimization. We briefly present two straight-forward applications to illustrate. The first application is the CAN [18] overlay network. We compare the overlay routing efficiency in terms of end-to-end delay of a 1000-host 3-dimensional CAN on a 100-node transit-sub topology [26] when using random overlay positions versus using network positions as overlay positions. The network positions are computed with the method above using 4 randomly chosen Landmarks among the nodes. We find that when using random overlay positions, the overlay delay is on av-

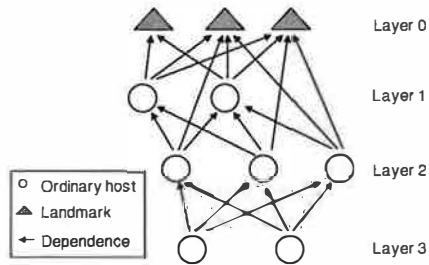


Figure 4: Hierarchical network positioning architecture.

and there exists no prior cached position information for the host. Landmarks and hosts periodically re-compute their positions to adapt to network topology changes.

3.3 Hierarchical Architecture

Position consistency is an important goal. To ensure that all host positions are consistent and have the same bases, a straight-forward solution is to use the positions of a set of Landmark hosts to define the bases and require all hosts to use the Landmarks as their reference points. This is the method outlined in Section 2. Although this approach guarantees consistency, it has some undesirable properties. As illustrated in Figure 1, *every* ordinary host probes *all* Landmarks to compute its coordinates. In such a system, the Landmarks and their network access links become performance bottlenecks, and there is no obvious solution to recover gracefully from Landmark failures.

Thus, we seek a decentralized solution that can at the same time maintain consistency. Our design is a hierarchical architecture. In this architecture, Landmarks still define the bases and can serve as reference points for hosts in the system. The main departure from the basic approach is that any host that has determined its position can be chosen by the membership server to be a reference point for other hosts. As a result, Landmarks become much less critical, and temporary Landmark failures will not halt the entire system. Currently, the membership server randomly chooses among the eligible hosts as reference points when the Landmarks are too heavily loaded or unavailable. However, to ensure consistency, we impose a hierarchical position dependency among the hosts. Figure 4 illustrates the hierarchical architecture. We say that host A *depends* on host B if A uses B as one of its reference points. We also define a notion called *layer number*. The layer number of a host is the maximum number of dependency hops separating it from the Landmarks. For convenience, the Landmarks are given layer number 0. We call a system that allows a highest layer number L an “ $L + 1$ layer system”.

By imposing a position dependency hierarchy, we ensure that the positions of all hosts have the same bases

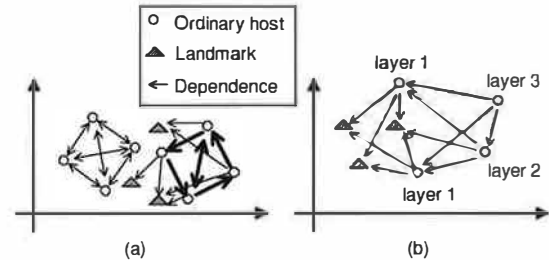


Figure 5: Host dependency in 2D Euclidean embedding. (a) Inconsistent. (b) Hierarchical, consistent.

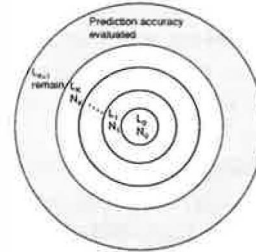


Figure 6: Logical host partitioning in layering experiment.

defined by the Landmarks. Note that if the position dependency is arbitrary, there can be a danger of losing consistency. In general, any dependency cycle can prevent hosts’ positions from converging since position updates are propagated through the cycle. An example is shown in Figure 5(a), where one cluster of hosts is partitioned, and hosts have cyclic dependencies. Thus the positions of the two clusters of hosts do not have the same bases, and some hosts’ positions may not converge.

The hierarchical dependency invariant is easy to maintain in the system by checking reference points’ layer numbers. Landmarks always have layer number 0. When a NPS daemon starts, it initializes its layer number to be the highest layer number allowed in the system, L_{max} (a static parameter retrieved from the membership server). To maintain the invariant, each host H with current layer number L_H simply refuses to depend on any reference point R that has a layer number $L_R \geq L_H$. When H accepts a set of reference points R_i , H updates its layer number to be $\max_i(L_{R_i}) + 1$. Figure 5(b) shows a 2-dimensional example.

3.3.1 How Many Layers?

An important question to ask is, how many layers are needed and how is accuracy affected? To shed light on these questions, we have performed idealized simulations on Internet-like synthetic topologies. Figure 6 illustrates the methodology. Given a set of hosts, we partition them into $K + 2$ different layers. Layer L_0 contains N_0

$N_{1..K} = 500$	$K = 0$	$K = 1$	$K = 2$	$K = 3$	$K = 4$
Inet 3050	.42	.46	.87	.76	1.01
Inet 6000	.35	.41	.69	.66	.83
PLRG 3000	.32	.44	.57	.61	.68
PLRG 6000	.35	.48	.81	.72	.97

Table 1: 90 percentile relative error ($N_0 = 40, N_{1..K} = 500$, varying K).

$K = 1$	$N_1 = 500$	$= 1000$	$= 1500$	$= 2000$
Inet 3050	.46	.45	.44	.44
Inet 6000	.41	.41	.42	.41
PLRG 3000	.44	.44	.43	.43
PLRG 6000	.48	.47	.46	.46

Table 2: 90 percentile relative error ($N_0 = 40, K = 1$, varying N_1).

Landmarks. For $i = 1, \dots, K$, N_i randomly chosen hosts are placed into layer L_i . The remaining hosts are placed in layer L_{K+1} . In each experiment, for $i = 1, \dots, K + 1$, each host in layer L_i randomly picks N_0 hosts in layer $L_i - 1$ as its reference points. After every host has been embedded, we evaluate the relative error of the distance predictions among the hosts in layer L_{K+1} . The relative error of the distance prediction between a pair of hosts is defined as:

$$\text{relative error} = \frac{|\text{actual} - \text{predicted}|}{\min(\text{actual}, \text{predicted})} \quad (4)$$

By varying K and N_i , we can study the impact of adding layers or changing the layer size on embedding accuracy. Note that these experiments are centrally simulated under static conditions, no network dynamics are introduced. We conduct these experiments on 4 different synthetic power-law networks. Two are generated by the Inet 3.0 generator [23] (3050 and 6000 nodes), two are generated based on the PLRG model [2] (using the largest connected component, roughly 3000 and 6000 nodes). As usual, nodes are randomly placed in a square region, and the delay of a link is the Euclidean distance between the end points of that link. End-to-end delay is the shortest path delay. For each network, we choose 5 random sets of 40 Landmarks that are well-separated. For each configuration, we run 5 experiments, one per each set of Landmarks. For embedding, an 8-dimensional Euclidean space is used. The resulting relative error among hosts in the outer-most layer is computed and the 90 percentile is reported.

Table 1 and Table 2 show the results. Although we present only the 90 percentile relative errors, note that the the 50 percentile relative error is less than 0.2 for all experiments. From Table 1, we see that with $K = 1$, the relative error is only slightly higher than $K = 0$ when every host uses the Landmarks as reference points.

However, allowing more layers inflate the relative error, a sign of error accumulation. Concentrating on $K = 1$, from Table 2, we see that the relative error appears to be independent of the size of layer 1.

Interestingly, we observe that the relative error of the distance predictions between hosts in layer 1 and layer 2 is better than that among layer 2 hosts. For $K = 1$ and $N_1 = 2000$, the 90 percentile relative error of the distance predictions between layer 1 and layer 2 hosts ranges from only 0.23 to 0.28. Thus, prediction accuracy across layers is still maintained.

3.3.2 The 3-Layer System

Based on our findings, we believe a 3-layer system ($K = 1$) is very promising. First of all, accuracy is only slightly affected, and secondly under reasonable assumptions and based on our practical experience with the system, a 3-layer system can already reach an enormous membership size. It is important to note that only a few RTT measurements are needed to perform a position computation, thus the bandwidth overhead is very low. Assuming a 12-hour re-computation cycle, each host in our system conservatively generates less than 0.5bps of measurement traffic to each reference point. Assuming layer 0 Landmarks have 1Mbps of dedicated bandwidth to support layer 1 hosts, layer 1 hosts dedicate 10kbps to support layer 2 hosts, and 20 reference nodes are used by each node, layer 1 can reach 2 million nodes, and layer 2 can already reach 2 billion nodes!

3.4 Ordinary Host Positioning

The ordinary host positioning method at a high level follows the one described in Section 2 except of course the reference points used by a host are not necessarily the Landmarks. However, several types of dynamics need to be considered when positioning a host in the system: (1) Variable queuing delay in a network path makes it unclear how many RTT samples are needed to decide the network distance (i.e. minimum RTT) has been measured. (2) When a host is updating its position, its reference points could also be simultaneously updating their positions. (3) Network topology change can increase the network distance of a path, so always keeping the minimum RTT will not detect this kind of topology change.

To automatically adapt to (1) and (2), we employ an iterative positioning procedure that terminates when a stability heuristic condition is met. More precisely, in each iteration, a host only requires one probe exchange with each of its reference points. This provides the host with one additional RTT sample for each reference point and the latest position of each reference point. The newest minimum RTT samples are used to update the network distances. Based on the latest network distances and ref-

erence point positions, a new embedding position is computed by minimizing Eq. 3. A random wait time of up to one second is introduced between each iteration. The heuristic is that if in three consecutive iterations the position of the host has moved by less than one millisecond in the Euclidean space, then the procedure is terminated and stability is declared. In our experience, by initially setting the position of a host at the origin, stability can typically be reached in under 15 seconds, including time spent in pacing network probes. To adapt to topology changes, re-positioning of a host starting from its current position will be performed periodically.

To address (3) during re-positioning, the strategy is to reconsider the current known minimum RTT of a path based on a set of new RTT measurements. When a host begins the iterative procedure to re-compute its position, it has the previously known minimum RTT d_{old} to each reference point. The host first obtains 10 new RTT samples to each reference point. Supposed the minimum of these 10 new samples is d_{new} . If d_{new} differs from d_{old} by less than one percent, then we use d_{old} as the initial network distance and begin the iterative procedure. Otherwise, we use d_{new} . As a result, if the network distance is increased by less than one percent, it will be ignored. A decrease in network distance can be discovered eventually in the iterative procedure.

3.5 Decentralized Landmark Positioning

In the basic network positioning method described in Section 2, Landmarks measure their inter-Landmark distances and ship all the data to a central node to compute Landmarks' positions in an Euclidean space that minimize $f_{obj1}(\cdot)$ (Eq.(1)). In contrast, NPS implements a decentralized method. Observe that $2 \times f_{obj1}(\cdot)$ can be expanded as:

$$2 \times f_{obj1}(c_{\mathcal{L}_1}, \dots, c_{\mathcal{L}_N}) = \sum_{i \neq 1} \mathcal{E}(d_{\mathcal{L}_i \mathcal{L}_1}, \hat{d}_{\mathcal{L}_i \mathcal{L}_1}) + \sum_{i \neq 2} \mathcal{E}(d_{\mathcal{L}_i \mathcal{L}_2}, \hat{d}_{\mathcal{L}_i \mathcal{L}_2}) + \dots + \sum_{i \neq N} \mathcal{E}(d_{\mathcal{L}_i \mathcal{L}_N}, \hat{d}_{\mathcal{L}_i \mathcal{L}_N})$$

Thus, the objective function can be decomposed into N terms, each term corresponds to how one Landmark relate to the others. We implement a decentralized strategy which in essence randomly chooses one term from the above equation, for example the term for \mathcal{L}_2 , and computes the corresponding host's position that locally minimizes that chosen term, and then repeats until no further progress can be made.

More specifically, all Landmarks upon boot-up are initially positioned at the origin of the Euclidean space. Each Landmark then uses all the other Landmarks as its reference points, probes the other Landmarks, retrieves their latest positions and obtains network distance samples, and computes a new position for itself. These steps are in fact very similar to what an ordinary host does when computing its position. A random pause time of up to one second is introduced after each computation step. Then the steps repeat until a convergence heuristic criterion is met. The criterion again is that convergence is achieved when in 3 consecutive iterations a Landmark's position has not moved by more than one millisecond in the Euclidean space. Note that all Landmarks need to roughly simultaneously begin re-considering their positions. We do this by triggering a Landmark to begin re-considering its position when it is probed by another Landmark, a signal that the other Landmark is re-considering its position. To make it harder for a spoofed Landmark to trigger computation, Landmarks exchange random 32-bit authorization IDs when they probe each other and a node must send the correct authorization ID to a Landmark to trigger it. In our experience, this approach can embed 20 Landmarks starting from their origin positions in approximately one minute and the resulting positions are just as accurate as the centralized approach. Again, to adapt to topology changes, the Landmarks also need to re-compute their positions periodically. If the network change is minimal, convergence can be achieved in a small number of iterations.

3.5.1 Maintaining Stability

The Landmarks' positions define the bases of the Euclidean space which all other hosts share. If the network topology is changed, then the Landmarks' position should be updated accordingly to reflect the change. However, if the network topology has not changed, it is important to minimize unnecessary drifting of the Landmarks' positions when they are re-computed. To increase the stability of the Landmarks' positions, we apply the following mechanisms.

First, to get good measures of the current network distances, before any position re-computation, a Landmark probes every other Landmark 50 times. In addition, piggybacked in these probes, the Landmarks conduct bi-directional exchanges of their current known minimum RTT to synchronize. This is important because two Landmarks must agree on the same network distance between them to reduce oscillations in positions. Once the probings are done, the new minimum RTTs are compared to the previous known minimum RTTs. Again if the change is less than one percent, the old value is kept as the initial network distance. Finally, if the network

distances from a Landmark to all other Landmarks have not been changed by more than one percent, the Landmark keeps its old position.

3.6 Congestion Control

Probing from ordinary hosts are congestion controlled in the system. This is to prevent overwhelming a reference point if too many hosts are simultaneously probing it. To implement congestion control, each probe packet carries a sequence number. By observing the sequence numbers in the probe reply packets, we can infer packet losses in the same manner as TCP. The initial probing rate is one probe per second in the implementation. An additive-increase-multiplicative-decrease (AIMD) [25] rate adjustment procedure is applied when probe reply packets are received or lost. The increase and decrease factors can be tuned to be less aggressive than TCP if desired, but this tuning is out of the scope of this paper. Probing rate is also upper bounded by a constant (the implementation uses 10 probes per second as maximum rate).

Note that probing between Landmarks are sent at a constant rate (10 probes per second) and no congestion control is applied. This is to ensure no Landmark will fall far behind in the position computation process.

3.7 Triggered Re-Positioning

Although hosts periodically re-compute their positions, the setting of the re-computation interval represents a trade-off between the load on the system and the accuracy of the hosts' positions. To be conservative, by default a host re-computes its position only once every 12 hours in the current system. However, we use triggered re-positioning to improve the accuracy and consistency of the system when network topology changes. When a reference point has undergone a drastic change in position indicating a large change in network topology, the reference point can trigger dependent hosts to re-compute their positions. The aggressiveness in triggering dependent hosts is controlled by the reference point based on the load it can tolerate. To ensure only a reference point of a host can trigger the host to re-compute its position, we do the following. Before beginning computing a new position, a host randomly chooses a 32-bit authorization ID. This ID is included in probe packets sent to the reference points. A reference point records the authorization ID for each dependent host. In the current system, when a reference point's position has moved by over 10 milliseconds in the Euclidean space, it triggers a dependent host to re-compute by sending the corresponding authorization ID to the dependent host. The dependent host re-computes its position if the authorization ID is valid.

Note that the Landmarks are never triggered by ordinary hosts in the system. They periodically re-compute their positions. The current period used in the implementation for Landmarks is 3 hours.

3.8 Detecting Malicious Reference Points

In our system, Landmark nodes are trusted entities, but ordinary hosts in the system that serve as reference points for other hosts could potentially lie to their dependents if they are malicious. They could lie about their actual positions and/or inflate the network distances by holding onto probe packets. We separately discuss two types of lies: (1) continuously changing lies, (2) fixed lies.

First, the damage of continuously changing lies is limited. This is because a reference point that continuously changes its position can easily be noticed and be eliminated by the dependent. We do this by putting a time limit on each reference point within which the reference point's position must stabilize or else it will be removed. A time limit is also put on the stabilization of the network distance (i.e. minimum RTT) to a reference point.

The more damaging lie is the fixed type in which a malicious reference point consistently reports the same false position and/or inflated network distance to each dependent. Such a malicious reference point would appear to be normal to a dependent and it would be difficult to find a mechanism that could 100% accurately identify such a lying reference point. One approach to coping with this type of lie is to eliminate a reference point if it fits poorly in the Euclidean space compared to the other reference points. The specific procedure in the system is as follows. Assume there are N reference points \mathcal{R}_i with positions $P_{\mathcal{R}_i}$, and the network distances from a host \mathcal{H} to them are $D_{\mathcal{R}_i}$. After \mathcal{H} computes a position $P_{\mathcal{H}}$ based on these N reference points, for each \mathcal{R}_i , it computes the fitting error $E_{\mathcal{R}_i}$ as $\frac{|distance(P_{\mathcal{H}}, P_{\mathcal{R}_i}) - D_{\mathcal{R}_i}|}{D_{\mathcal{R}_i}}$. Then we decide whether to eliminate the reference point with the largest $E_{\mathcal{R}_i}$. The current criterion is that if $max_i(E_{\mathcal{R}_i}) > 0.01$ and $max_i(E_{\mathcal{R}_i}) > C \times median_i(E_{\mathcal{R}_i})$, where C is a constant, then the reference point with $max_i(E_{\mathcal{R}_i})$ is eliminated. That is, a reference node is rejected if it has fitting error significantly larger than the median error among all reference nodes. We use median error as the criterion since it is more robust than the average error and cannot be easily affected by a minority of malicious nodes. In Section 5, we explore the choice of the constant C .

4 NPS Implementation

The system is implemented on the Linux platform. It includes two components: (1) the NPS membership server, (2) the NPS daemon. The membership server's main

tasks are to provide basic system configuration information and to serve as a rendezvous point for NPS daemon coordination. Since the membership server does not need to synchronize dynamically changing information, it can be replicated easily. All network communications in the system are implemented with UDP datagrams.

4.1 NPS Membership Server

The key functions of the membership server are to (1) identify the Landmarks, (2) provide initial configuration parameters to NPS daemons (i.e. whether the daemon is a Landmark, the number of reference points a daemon should use, the geometric space used for embedding, and the maximum number of layers allowed in the system), (3) maintain a subset of the current hosts that can potentially serve as reference points, (4) hand out potential reference points when requested, and (5) control the number of hosts in each layer. The membership server is an event-driven process. The global parameters are read from a text configuration file.

When a NPS daemon is started, it first asks the membership server for a set of initial configuration parameters (i.e. item (2) above). If the NPS daemon is identified as a Landmark, the number of reference points to use is one less than the number of Landmarks since the other Landmarks are its reference points. Otherwise, the number of reference points to use is the number of Landmarks.²

When a NPS daemon needs reference points to perform an embedding computation, it contacts the membership server. If the NPS daemon is a Landmark, the other Landmarks are always returned as the reference points. Otherwise, the membership server has some flexibility in giving out reference points. If the NPS daemon reports it is currently in layer L , the membership server will avoid giving out any reference points in layer L or larger because those reference points will be rejected. Note that a NPS daemon assumes it is in the highest layer allowed in the system when it starts. Note also that a NPS daemon will verify the layer number of its reference points during probing and thus the layer number information maintained by the membership server does not need to be precise. The membership server can also select reference points to roughly adjust the number of hosts in each layer. For example, it can maintain a limited number of hosts in layer 1 to ensure that the Landmarks are not overloaded by measurement traffic. Once that threshold is reached, it can direct other hosts to layer 2 or higher by giving them reference points in layer 1 or above.

When a NPS daemon has successfully computed a stable embedding position, it reports to the membership server its stable status and its current layer number. This

²It is possible to use fewer reference points, but this variation is not considered in this study.

information is kept as soft state at the membership server. Stale entries are periodically flushed from memory. The network load on the membership server is quite minimal. In the experiments, a newly started NPS daemon altogether sends and receives less than 600 bytes of data between itself and the membership server to boot up, locate reference points, and report its status. If necessary, multiple membership servers can be deployed.

4.2 NPS Daemon

The purpose of the NPS daemon is to compute and maintain the embedding position of the host which may be a Landmark or an ordinary host, and cope with the dynamics in the network or the system. The NPS daemon consists of two processes. The parent process implements the NPS logics and protocols, and the child process is a computation engine that solves the optimization problems in embedding computations using the simplex downhill algorithm [14]. The two processes are both event-driven and communicate via a full duplex pipe. The concurrency is desirable because we want the parent NPS process to remain responsive to various events while an optimization problem is being solved (which could last for a few milliseconds).

Once a NPS daemon has configured itself with parameters obtained from the membership server, it begins to compute and maintain its embedding position. At the beginning of an embedding computation cycle, a NPS daemon first declares itself unstable (the precise definition of stability is discussed in Section 3.4). It then makes sure it knows a sufficient number of valid reference points, if not, it asks the membership server for more. The network distance between itself and each reference point is measured by timing the delay between the sending of a probe message and the receiving of a reply from the reference point. When a reference point replies to a probe message, it piggybacks its current coordinates and its current layer number onto the message. As soon as a network distance estimate is obtained for each valid reference point, the NPS daemon performs the embedding computation. This process is repeated until the stability criterion is met. The stable coordinates are stored on disk so that they can be retrieved after a daemon restart. The current layer number is then reported to the membership server, and a re-computation is scheduled. This completes one embedding computation cycle. If 12 probes are sent from an ordinary host to a reference point without any reply, the reference point is removed and a new one is obtained from the membership server. Sequences of probes to different reference points are started at least 10ms apart. The selection of these constants are not significant, they just seem reasonable.

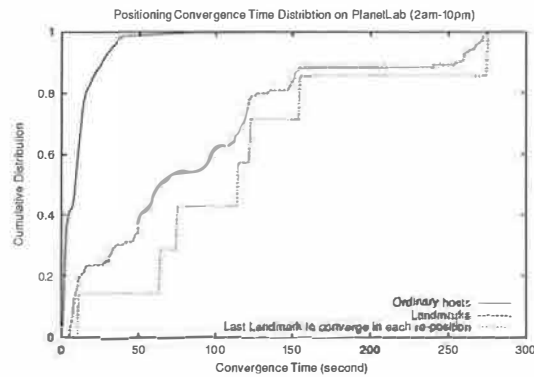


Figure 7: Positioning convergence time distributions.

4.2.1 NPS Application Programming Interface

The NPS daemon currently supports a simple application programming interface for network position queries. An application can query the NPS daemon by sending it a UDP query message. The NPS daemon replies with the dimensionality of the Euclidean space and current numerical coordinates of the host in a UDP reply message. Currently the coordinates are represented as 32-bit integers in unit of microsecond.

4.2.2 Working with NAT Hosts

Network Address Translation (NAT) [21] is becoming an increasingly ubiquitous solution to incrementally scale up the size of the Internet. However, a problem with NAT is that it can prevent in-bound data connections. The system has been carefully designed to work with NAT. First, the system uses only UDP datagrams for communication, and a NPS daemon is identified at a membership server by the IP address and UDP port number carried in the messages it sends. Thus, NPS daemons behind NAT can be uniquely identified by the membership server. In addition, the messages between a NPS daemon behind NAT and the membership server would establish forwarding state in the NPS daemon host's NAT gateway. Under typical implementations of NAT, another NPS daemon can communicate with the NPS daemon behind NAT using this same NAT forwarding state. Thus the system can work around the in-bound connection problem.

5 Experiments

In this section, we present experience with the system on PlanetLab. We also present results from controlled experiments using a simulator.

5.1 PlanetLab Experience

The NPS system is operational on PlanetLab. Here, we report our experience from a particular 20-hour period of

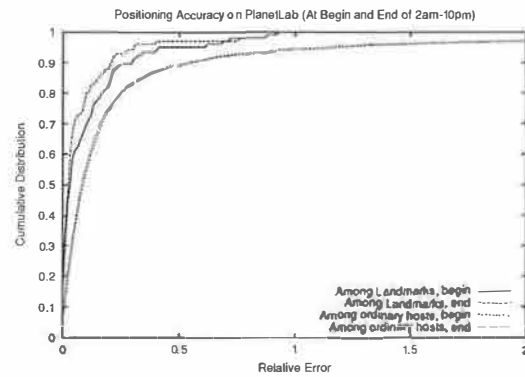


Figure 8: Positioning accuracy on PlanetLab.

operation of the system on a weekday from 2am to 10pm. Note that the PlanetLab is a shared test-bed.

We use 127 PlanetLab nodes. We first collect the 127×127 network distances using ping (100 RTT samples per path), then we apply clustering to select 15 well-distributed nodes as Landmarks. The remaining 112 hosts serve as ordinary hosts. A membership server is set up at our institution. The Euclidean space is configured to have 8 dimensions. All the ordinary hosts use the Landmarks as reference points unless Landmarks are down. The NPS daemons on Landmarks are started 5 minutes prior to the starts of ordinary hosts. Landmarks update their positions once every 3 hours. Ordinary hosts by default update their positions once every 12 hours, but can also be triggered by a Landmark to update if the Landmark has moved its position by more than 10 milliseconds. Probing congestion control is also enabled.

5.1.1 Convergence Time

Figure 7 shows the positioning convergence time distributions during this 20-hour period. The convergence time is measured starting from the first probe to any reference point is sent by a host until the host's position has not changed by more than one millisecond over 3 consecutive computation iterations. Note that most of this time is spent in pacing network probes to reference points. First, consider the distribution for ordinary hosts, we can see that updating the position of an ordinary host takes less than 15 seconds in 80% of the cases. As expected, however, the convergence times for Landmarks are generally longer since Landmarks simultaneously updates their positions and need more time to agree on their positions distributedly. We have also computed the time it takes the last Landmark to converge in each of the 7 update rounds during the 20-hour period. We can see that in most rounds, all Landmarks converge in less than 160 seconds. In a better provisioned environment, we expect the convergence times to be reduced.

The convergence times we observe are far shorter than the time scale at which the Internet topology changes, which is typically on the order of a day [27], thus the system is sufficiently agile. When the NPS daemon is run as a background process, the only occasion an application asking for the position of a host needs to wait for the NPS daemon to converge is when the NPS daemon is initially started without any prior position information. In most cases, applications can obtain the position information instantaneously.

5.1.2 Position Accuracy Over Time

Figure 8 summarizes the accuracy of the system on PlanetLab. The accuracy metric is relative error (Eq.(4)). We have plotted the cumulative distribution of relative error for different classes of hosts near the beginning of the 20-hour period and near the end of the 20-hour period. First, since Landmarks directly use the inter-Landmark distances in computing positions, not surprisingly, the resulting Landmark positions approximate the distances among Landmarks well, achieving a 90 percentile relative error of roughly 0.25. Comparing the accuracy of the Landmarks' positions at the beginning and end of the 20-hour period, we see that the level of accuracy is maintained by the system without degradation. This is an important validation of the system's ability to maintain position accuracy over time in a dynamic environment.

For the ordinary hosts, we consider only the position accuracy for the inter-ordinary-host distances. Note that these distances are not used in computing the ordinary hosts' positions. The level of accuracy achieved by the system is very good with a 50 percentile relative error of 0.08 and a 90 percentile relative error of 0.52. This accuracy distribution is very similar to previous network measurement-based results reported in [15] and [22]. The point that we wish to again emphasize is that the level of accuracy is maintained by the system over time without degradation.

5.2 Controlled Experiments

To explore specific aspects of the system, we perform controlled experiments of the system in a simulator. This allows us to quantify the impact of various designs of the system under different scenarios, and to allow us to compare how close the system's accuracy is to the target accuracy of an idealized simulation.

Our network simulator is event-driven. It implements a set of system interfaces to interact with NPS components. The important interfaces are `get_current_time()`, `set_timer()`, `unset_timer()`, `send_message()`, and `solve_for_coordinates()` to compute the coordinates for the requester. Only one computation engine exists in the simulator, the real computation time is recorded and

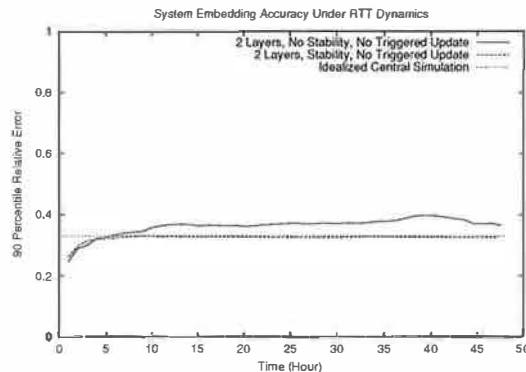


Figure 9: 2 layer system embedding accuracy under RTT dynamics.

is used to simulate the computation delay. The NPS daemon and membership server provide an `event_handler()` interface to receive either timer events, messages, or computation results.

The simulator is capable of simulating system dynamics. The underlying network topology can be changed during simulation. It can also simulate variations in the network round-trip times. We do this by adding to a network delay d some exponentially distributed noise with a mean m , and we choose $0 < m \leq d$. Our goal is only to observe how different aspects of the system function under noise, and we do not claim that noise generated by this method is representative of the noise on the Internet. It has been observed that Internet RTT measurements are often fairly close to the minimum RTT [1]. In the simulator, 12% of the paths are given the worst case variation where we set $m = d$. Intuitively, in this case only one in 100 RTT measurements is within 1% of the true minimum. For 50% of the paths, $m > 0.05d$, and for 10% of the paths, $m < 0.005d$.

We use a 1044-node PLRG [2] topology for the simulations. As usual, nodes are randomly placed in a square region, and the delay of a link is the Euclidean distance between the end points of that link. End-to-end delay is the shortest path delay. First, a set of 20 well-separated Landmarks are randomly chosen. We use an 8-dimensional Euclidean space for the embedding. To generate an ideal accuracy target, we conduct an idealized simulation of networking positioning. We use the 90 percentile relative error as a point to compare between different experiments. We find that using the 20 Landmarks as reference points for the entire system, the 90 percentile relative error achieved is 0.33. This can be considered a rough lower bound for the 90 percentile relative error for the system. The goal will be to achieve an accuracy as close to this possible. In the system simulations, the Landmarks recompute their positions once every 3 hours.

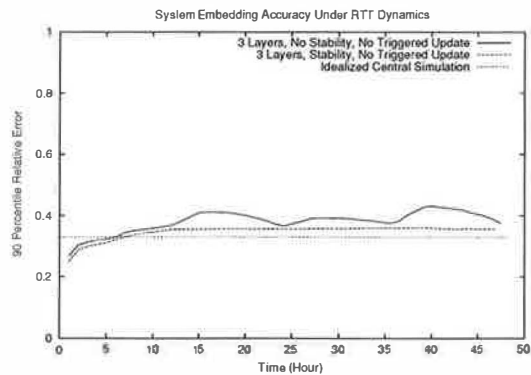


Figure 10: 3 layer system embedding accuracy under RTT dynamics.

The ordinary hosts, on the other hand recompute their positions once every 12 hours. Triggered re-positioning are turned on in some experiments. The Landmarks recompute more frequently so that any topology change can be fairly promptly detected by the Landmarks. The experiments simulate the life of the system for 48 hours. The Landmarks begin the process of computing their positions at time zero. Unless otherwise stated, the 1024 ordinary hosts join the system in random order, starting at the 10 minute mark, with an exponentially distributed inter-arrival time chosen such that the last host joins the system at roughly the 12 hour mark. This spreading is done to expose the embedding accuracy of the system under asynchronous host computations. To summarize the performance of the system over time, at every hour in the simulation, we compute the 90 percentile relative error for all the network distances among hosts that are in the system at the moment.

5.2.1 System Accuracy and Effectiveness of Stability Control

We first present the result of an experiment under RTT dynamics (no real topology change) where all hosts use the Landmarks as their reference points (i.e. a 2 layer system). In this experiment, the triggered re-positioning mechanism is turned off. The stability control mechanisms for Landmarks is experimented with. The results are shown in Figure 9. The accuracy of the idealized simulation is also plotted as a baseline for comparison.

First, note that without the stability control mechanisms described in Section 3.5.1, we can clearly see that the accuracy of the system is adversely affected by the drifting of Landmarks' positions. The problem is that as the Landmarks' positions drift every 3 hours, different ordinary hosts are computing their positions based on different sets of Landmark positions without knowing it, which leads to a noticeably higher position error. With

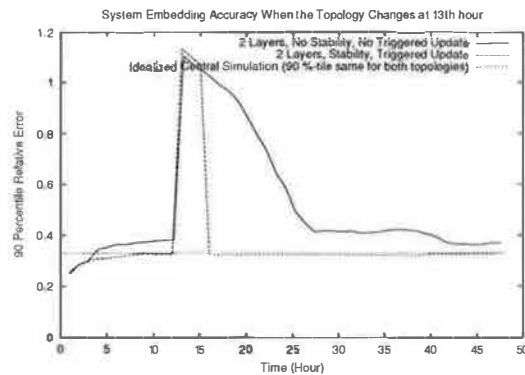


Figure 11: 2 layer system embedding accuracy under drastic topology change.

the stability control mechanisms, the system's accuracy is stable throughout the 48 hour period. Second, observe that the system's accuracy is matching that of the idealized simulation. The lower error at the beginning is because hosts are still joining in the dynamic system. Thus, the system achieves the basic objective of matching the performance of an idealized simulation.

In the simulated environment, an ordinary host on average takes 14.8 seconds to converge to a position. The last Landmark to converge in each round takes on average 41.2 seconds. These figures are not so different from those we see on PlanetLab.

Figure 10 shows the result when only 300 hosts are allowed to be at layer 1 and the other 724 hosts are forced to be at layer 2 and use layer 1 hosts as reference points (i.e. a 3 layer system). As expected, there is a small accuracy penalty over the 2 layer case. However, what is gained here is a much larger potential system size and robustness since hosts do not have to use Landmarks as reference points. We also observe that the impact of not having the stability mechanisms is magnified by the decentralization (compare to Figure 9). With the stability mechanisms, the accuracy of the system remains stable.

5.2.2 Effectiveness of Triggered Re-Positioning

In the next set of experiments, we keep the settings the same as before, except that at the 13th hour into the simulation, we replace the underlying network topology with a different PLRG. This is done to simulate a drastic and instantaneous change in the network topology. What we want to show is that the NPS system can gracefully adapt to such catastrophic network change and eventually converge to a consistent global embedding. Coincidentally, the 90 percentile relative error achievable in an idealized simulation for the second topology is also 0.33. Figure 11 shows the result for a 2 layer system.

First, notice the sharp jump in relative error at the 13th

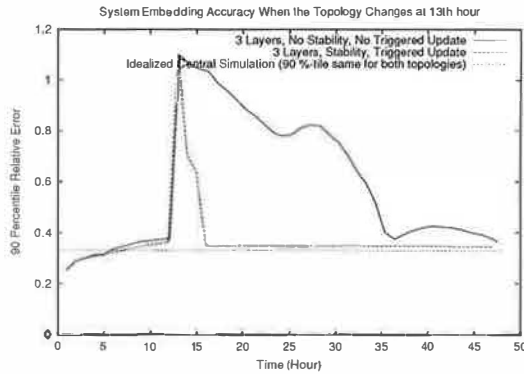


Figure 12: 3 layer system embedding accuracy under drastic topology change.

hour due to the topology change. Over time, the system is able to detect the topology change and adapt. We can also see the benefits of triggered re-positioning and the stability mechanisms. Without either of these features, the system adapts but at a slow pace since hosts do not recompute their positions for 12 hours, and the system accuracy still fluctuates after 12 hours due to instability. With both features, we see that after the 15th hour when Landmarks have recomputed their positions, other hosts are triggered to update their positions. By the 16th hour, all hosts have reached their new positions, the accuracy of the system is stable and matches that of the ideal case.

Figure 12 presents the result for the same network topology change scenario but with a 3 layer system where 300 hosts are in layer 1. First, consider the case with no triggered re-positioning and stability mechanisms. Because in the 3 layer system there is one level of indirection between the majority of the hosts and the Landmarks, it takes roughly one 12 hour period for all the layer 1 hosts to all move to their new positions, and roughly another 12 hour period for all the layer 2 hosts to move to their new positions relative to the layer 1 hosts. In contrast, with the stability and triggered update features, convergence to good performance happens quickly even with a layer of indirection. The layer 1 hosts are able to inform layer 2 hosts of the network topology change and within 1 hour after the Landmarks detect the topology change, all hosts have converged to their new positions with high and stable accuracy.

5.2.3 Effectiveness of Malicious Reference Node Detection

To see the effects of lying reference points, we conduct several experiments with a 3 layer system with 300 hosts in layer 1. First 10% of the ordinary hosts are designated to be malicious. Each of them randomly chooses a number between 0 and 1000. It then consistently lies

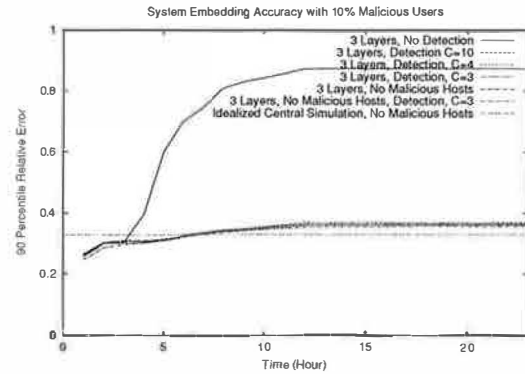


Figure 13: 3 layer system accuracy with 10% lying hosts.

about its position and inflate network distances by that amount (in unit of millisecond). In Figure 13, we show the effect of these malicious hosts to the system in a 24 hour simulation period. When the malicious reference node detection mechanism is off, notice the sharp climb in relative error when layer 2 hosts begin to join the system. Without any malicious reference node detection mechanism, the accuracy of the system is destroyed by the lying layer 1 hosts. Fortunately, the malicious reference node detection mechanism is able to dramatically restore performance to nearly the same level as the “3 layers, no malicious hosts” case. That is, the mechanism is highly effective at detecting malicious hosts. We have experimented with several different sensitivity constant C values to tune the aggressiveness of the mechanism and found that a conservative value of 4 is sufficient. Moreover, when there is no malicious hosts in the system, the malicious reference node detection mechanism only slightly affects the system’s overall accuracy.

6 Related Work

This work is aimed at building a networking positioning system that can be used by Internet applications. As such, this work deals broadly with issues including system architecture, algorithms design, implementation details, system tuning, and to a limited extent security. To the best of our knowledge, this is the first study to consider system-building issues in network positioning. We have identified several key system issues such as position consistency, stability, and adaptivity that are important in a practical system. While this system is being developed, many parallel work has explored a variety of issues in network positioning.

In [20], an optimization method called Big-Bang-Simulation (BBS) is proposed. This method simulates the error in an embedding as force fields. It uses a multi-phase procedure to reduce the error in the embedding iteratively. It has been shown that this method is compu-

tationally efficient and has slightly better relative error performance than the downhill simplex minimization algorithm used in our system. A distributed version of BBS can potentially be developed and used in NPS.

In [16], a distributed network positioning architecture called Lighthouse and a different positioning mathematical framework based on the Gram-Schmidt process rather than multi-dimensional optimization are proposed. In the Lighthouse architecture random hosts with computed positions are used to serve as reference points in the system to enhance scalability, and hosts that use different reference points as bases correct for the difference by using a position translation matrix. The PIC [5] study in parallel proposes an architecture similar to that of Lighthouse, but uses a multi-dimensional optimization framework for its robustness. In addition, the PIC study explores reference point selection algorithms, as well as proposes a triangle inequality based criterion for detecting malicious reference nodes. Interestingly, the PIC malicious reference nodes detection mechanism works somewhat similarly to our embedding error based approach. Comparing these distributed architectures to the hierarchical architecture in NPS, the position dependency structure is not controlled in Lighthouse or PIC, thus consistency becomes a concern in these designs. Our work focuses on system issues and does not consider reference node selection algorithms.

In [6], a highly symmetric, distributed network positioning architecture called Vivaldi is proposed. In this architecture, hosts do not need to have computed positions before serving as reference nodes. Instead, every node starts at the origin position and continuously measures network distances against a small set of random reference nodes and update its position to minimize the locally observed embedding error. The accuracy of this approach turns out to be very promising compared to GNP. A significant property of this approach is that in the steady state, the positions of all hosts are continuously evolving and consistent dependency is not ensured. It remains to be shown how quickly positions change in this architecture, how frequent are re-computations required to maintain accuracy, and how quickly this approach adapts to network topology changes.

A recent study [22] has shown the potential of using Lipschitz embedding with dimensionality reduction based on principal component analysis (PCA) as a method for network positioning. Moreover, in this method, it is easy to introduce multiple Landmark sets for hosts to use to increase scalability and use the translation matrices between the different sets of Landmarks to correct for the difference in bases accordingly. This study also empirically examines the intrinsic dimensionality in large network distance data sets and found the

dimensionality to be low. Compared to Euclidean embedding, Lipschitz embedding is much more efficient to compute and the accuracy is comparable to Euclidean embedding. A parallel study [13] has also suggested the use of PCA of Lipschitz embedding to compute network positions. This study analyzes the difficulty in the non-linear optimization of Euclidean embedding, and show that a PCA based scheme is more computationally efficient. This study also explores methods to reduce the number of Landmarks that need to be probed without adversely affecting the accuracy. Distributed versions of these methods however remain to be developed.

Besides network positioning, an alternative way to provide network topology information to applications is to supply them with network distance estimates directly. In particular, the IDMaps [7] service provides network distance information. IDMaps builds a simplified overlay topology map of the Internet based on network measurements performed by Tracer nodes in the network. Distance predictions are then computed by performing shortest path routing on this topology map model. IDMaps supports a general distance query interface such that an application can query IDMaps servers to find out the network distance between two hosts. Comparing to the IDMaps service, NPS is different in that the infrastructure nodes (Landmarks and membership servers) merely enables end hosts to use their own resources to compute their positions in the Internet and does not directly interact with any applications. It is up to the applications running on end hosts to decide how to use the computed locations. Distance prediction for example can be computed by end hosts and is a by-product of the position information. The Isobar [4] project proposes an efficient overlay network delay monitoring mechanism that uses clustering techniques to group together hosts that have similar network distance observations and thus reduce the amount of monitoring traffic significantly.

There are also other indirect methods for determining nearby neighbors in the Internet. For example, nearby Internet hosts can be clustered implicitly based on Internet routing table information [11]. In [19], a Landmark distance vector based scheme called Binning is proposed to estimate the proximity between hosts. With such a scheme, the location of a host would be represented by the Landmark distance vector. This scheme however is not aimed at producing actual network distance estimates. Triangulated heuristic [8] is another solution, where Landmark vectors are also used as locations, and upper and lower bounds on the network distance between two locations are estimated. In [10], a scheme called Beaconing is proposed to find nearby network hosts. The idea is to use the distance from a host to a Beacon to determine the subset of hosts that lie within

a similar distance from the Beacon. By intersecting the subsets of hosts provided by multiple Beacons, a set of nearby hosts can be found.

7 Conclusion

In a very short time, network positioning has developed into a fascinating research area. This paper, to the best of our knowledge, is the first to study the system-building issues in network positioning. We have identified key issues such as consistency, adaptivity, and stability in building a network positioning system, and found that with a carefully designed system, these issues can be addressed effectively in practice. There has been a lot of interest in the research community in having access to a publicly available network positioning system, and we believe our prototype can be a first step in providing such a capability. The operational experience will guide the future evolution of the system. Ultimately, our goal is to provide a network positioning capability to all hosts in the Internet.

Acknowledgement

We thank our shepherd Honesty Young and the anonymous reviewers for their constructive feedbacks on this work. We also thank Sylvia Ratnasamy for providing us with the content-addressable network (CAN) software for experimentations.

References

- [1] A. Acharya and J. Saltz. A Study of Internet Round-Trip Delay. Technical Report CS-TR-3736, University of Maryland, College Park, 1996.
- [2] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of ACM Symposium on Theory of Computing*, pages 171–180, 2000.
- [3] S. Banerjee, Z. Xu, S.-J. Lee, and C. Tang. Service multicast for media distribution networks. In *IEEE Workshop on Internet Applications (WIAPP)*, San Jose, CA, June 2003.
- [4] Y. Chen, C. Overton, and R. H. Katz. Internet Iso-bar: A scalable overlay distance monitoring system. *Journal of Computer Resource Management, Computer Measurement Group, Spring Edition*, 2002.
- [5] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical Internet coordinates for distance estimation. Technical Report MSR-TR-2003-53, Microsoft Research, September 2003.
- [6] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. In *Proceedings of HotNets-II*, Cambridge, MA, November 2003.
- [7] P. Francis, S. Jamin, V. Paxson, L. Zhang, D.F. Gryniwicz, and Y. Jin. An architecture for a global Internet host distance estimation service. In *Proceedings of IEEE INFOCOM '99*, New York, NY, March 1999.
- [8] S.M. Hotz. Routing information organization to support scalable interdomain routing with heterogeneous path requirements, 1994. Ph.D. Thesis (draft), University of Southern California.
- [9] A.-C. Huang and P. Steenkiste. Network-sensitive service discovery. In *Proceedings of USENIX - USITS*, 2003.
- [10] C. Kommareddy, N. Shankar, and B. Bhattacharjee. Finding close friends on the Internet. In *Proceedings of IEEE ICNP*, Riverside, CA, November 2001.
- [11] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August 2000.
- [12] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August 2000.
- [13] H. Lim, J. Hou, and C.-H. Choi. Constructing internet coordinate system based on delay measurement. In *Proceedings of Internet Measurement Conference*, Miami, FL, October 2003.
- [14] J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.
- [15] T. S. E. Ng and H. Zhang. Predicting Internet networking distance with coordinates-based approaches. In *Proceedings of IEEE INFOCOM*, June 2002.
- [16] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouses for scalable distributed location. In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.
- [17] PlanetLab. <http://www.planet-lab.org>.
- [18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, 2001.
- [19] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proceedings of IEEE INFOCOM*, New York, NY, 2002.
- [20] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. In *Proceedings of IEEE INFOCOM*, San Francisco, CA, March 2003.
- [21] P. Srisuresh and K. Egevang. Traditional IP network address translator (Traditional NAT), January 2001. RFC-3022.
- [22] L. Tang and M. Crovella. Virtual landmarks for the internet. In *Proceedings of Internet Measurement Conference*, Miami, FL, October 2003.
- [23] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical Report UM-CSE-TR-456-02, University of Michigan, 2002.
- [24] Z. Xu, C. Tang, S. Banerjee, and S.-J. Lee. Receiver initiated just-in-time tree adaptation for rich media distribution. In *Proceedings of NOSSDAV*, Monterey, CA, June 2003.
- [25] Y. Yang and S. Lam. General AIMD congestion control. Technical Report TR-200009, Dept. of Computer Science, University of Texas at Austin, May 2000.
- [26] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE INFOCOM*, March 1996.
- [27] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: Routing, loss, and throughput. Technical report, ACIRI, May 2000.

Early Experience with an Internet Broadcast System Based on Overlay Multicast

Yang-hua Chu[†], Aditya Ganjam[†], T. S. Eugene Ng[‡], Sanjay G. Rao[†],
Kunwadee Sripanidkulchai[†], Jibin Zhan[†], and Hui Zhang[†]

[†]*Carnegie Mellon University* [‡]*Rice University*

Abstract

In this paper, we report on experience in building and deploying an operational Internet broadcast system based on Overlay Multicast. In over a year, the system has been providing a cost-effective alternative for Internet broadcast, used by over 4000 users spread across multiple continents in home, academic and commercial environments. Technical conferences and special interest groups are the early adopters. Our experience confirms that Overlay Multicast can be easily deployed and can provide reasonably good application performance. The experience has led us to identify first-order issues that are guiding our future efforts and are of importance to any Overlay Multicast protocol or system. Our key contributions are (i) enabling a real Overlay Multicast application and strengthening the case for overlays as a viable architecture for enabling group communication applications on the Internet, (ii) the details in engineering and operating a fully functional streaming system, addressing a wide range of real-world issues that are not typically considered in protocol design studies, and (iii) the data, analysis methodology, and experience that we are able to report given our unique standpoint.

1 Introduction

The vision of enabling live video broadcast as a common Internet utility in a manner that any publisher can broadcast content to any set of receivers has been driving the research agenda in the networking community for over a decade. The high cost of bandwidth required for server-based solutions or content delivery networks, and the sparse deployment of IP Multicast are two main factors that have limited broadcasting to only a subset of Internet content publishers such as large news organizations. There remains a need for cost-effective technology for low-budget content publishers such as broadcasters of seminars, workshops and special interest groups.

Recent work in Overlay Multicast [13, 9, 17, 7, 19, 28, 37, 20, 32, 23, 39, 10, 5] has made the case that overlay networks are a promising architecture to enable quick deployment of multicast functionality on the Internet. In such an architecture, application end-points self-organize into an overlay structure and data is distributed along the links of the overlay. The responsibilities and cost of providing bandwidth is shared amongst the application end-points, reducing the burden at the content publisher. The ability for users to receive content that they would otherwise not have access to provides a natural incentive for them to contribute resources to the system.

Most of the existing work, including our own earlier work [9, 8], focus on issues related to “protocol design,” and evaluate their potential using simulation or university-based Internet test-beds. We believe that an equally important and complementary style of research can be conducted using an “application-centric” approach. In this approach, the experience gained from the wide-spread operational use of an application by real users sets the direction for further research. The more content publishers and receivers rely on the application, the stronger the case for Overlay Multicast, validating its relevance as a research question. In addition, the unique experience obtained in the process leads to important insight that can motivate future research in the area.

In adopting the “application-centric” approach, our primary consideration was to provide a useful and deployable tool to the general public, and reach operational status as quickly as possible. Therefore, we identify and address a wide range of issues, some of which are not typically considered in protocol design studies, but affect the successful deployment of Overlay Multicast. Our system copes with dynamics in user participation, adapts to application performance and Internet dynamics, supports users that have a wide range of network bandwidth and supports users behind network address translators (NATs) and firewalls. We have built supporting mechanisms such as logging receiver performance, monitoring of system components, and recovering from component failures. In engineering our system, we have adopted simple or natural solutions, with the provision that the design decisions could be revisited in the light of future experience. This approach has accelerated the deployment of the system, and, consequently has led to faster feedback from real deployment.

The challenges involved in obtaining the operational experience we report in this paper must not be underestimated. First, we have invested significant effort in convincing content publishers and event organizers that it is worth their while to experiment with the new technology. Second, while we have made earnest efforts to get our system deployed, the participation of viewers in our broadcasts depends on a range of factors not under our control, including the content we have access to. Third, unlike conventional research experiments, we have frequently had to work under the pressure to succeed in even our earliest broadcast attempts. Failures would significantly deter event organizers and limit future adoption of our system. One consequence is that it is critical to adopt robust, stable and well-tested code – a performance refinement that

may seem trivial to incorporate may take months to actually be deployed.

In over a year, we have been building an operational broadcast system based on Overlay Multicast and deploying it among more than 4000 real users in real Internet environments for over 20 events. We view the design and deployment effort as an ongoing process, and report on the experience accumulated so far. Overall, our experience confirms that Overlay Multicast is easy to deploy and can provide reasonably good application performance. In addition, we believe that our unique set of data, analysis methodology, and experience are useful to the research community.

The rest of this paper is organized as follows. In § 2, we present an overview of the system. § 3, 4, and 5 presents the deployment experience, analysis methodology, and performance analysis of our system. § 6 presents key design lessons learned from the experience that are guiding the future research directions.

2 System Overview

Figure 1 gives a high-level overview of our broadcast system. The encoder takes the multimedia signal from the camera, converts into audio and video streams, and sends to the broadcast source. The broadcast source and receivers run an overlay multicast protocol to disseminate the streams along the overlay. Each receiver gets the broadcast stream, and forwards to the media player running on the same machine. In addition, the participating hosts send performance statistics to the monitor and log server for both on-line and post-mortem analyses.

The detailed software architecture at the source and the receiver is depicted in Figure 2. Tracing the data flow, the broadcast source encodes the media signal into audio and multiple video packet streams (a), marks the packets with priority bits (b), and sends them to the overlay modules (shaded blocks). Multiple streams and prioritization are discussed in § 2.2. The overlay modules replicate packets to all of its children (c). Packets are translated from Overlay ID (OID) to IP addresses (d), and forwarded to each child using prioritization semantics (e). Once a child receives packets, it translates IP addresses back to OIDs (1), selects the best video stream, adjusts the RTP/RTCP headers (2), and forwards to the media player (3). The use of OID is described in § 2.4. The child also sends each data packet to the overlay module which forwards the data to its descendants. The rest of this section describes each of these blocks in detail.

2.1 Overlay Protocol

We provide a sketch of the overlay protocol below as a basis for the rest of the discussion. Because our application is single-source, the protocol builds and maintains an overlay tree in a distributed fashion. The tree is optimized primarily for bandwidth, and secondarily for delay. Each node also maintains a degree bound of the maximum number of children to accept.

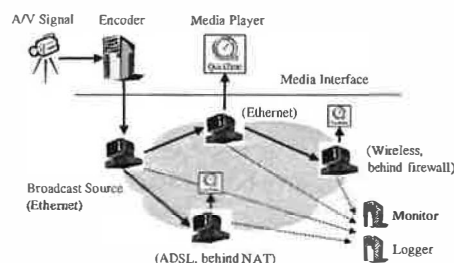


Figure 1: Broadcast system overview.

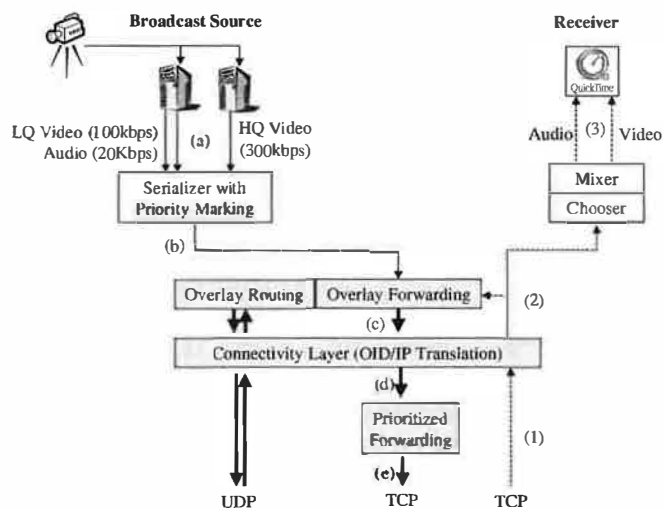


Figure 2: Block diagram of the software architecture for the broadcast source (left) and the receiver (right). Shaded blocks are shared by all hosts. Arrows indicate data flow.

Group Management: New hosts join the broadcast by contacting the source and retrieving a random list of hosts that are currently in the group. It then selects one of these members as its parent using the parent selection algorithm. Each member maintains a partial list of members, including the hosts on the path from the source and a random set of members, which can help if all members on the path are saturated. To learn about members, we use a gossip protocol adapted from [30]. Each host *A* periodically (every 2 seconds) picks one member (say *B*) at random, and sends *B* a subset of group members (8 members) that *A* knows, along with the last timestamp it has heard for each member. When *B* receives a membership message, it updates its list of known members. Finally, members are deleted if its state has not been refreshed in a period (5 minutes).

Handling Group Membership Dynamics: Dealing with graceful member leave is fairly straight-forward: hosts continue forwarding data for a short period (5 seconds), while its children look for new parents using the parent selection method described below. This serves to minimize disruptions to the overlay. Hosts also send periodic control packets to their children to indicate live-ness.

Performance-Aware Adaptation: We consider three dynamic network metrics: available bandwidth, latency and loss. There are two main components to this adaptation pro-

cess: (i) detecting poor performance from the current parent, or identifying that a host must switch parents, and (ii) choosing a new parent, which is discussed in the *parent selection* algorithm.

Each host maintains the application-level throughput it is receiving in a recent time window. If its performance is significantly below the source rate (less than 90% in our implementation), then it enters the probe phase to select a new parent. While our initial implementation did not consider loss rate as a metric, we found it necessary to deal with variable-bit-rate streams, as dips in the source rate would cause receivers to falsely assume a dip in performance and react unnecessarily. Thus, our solution avoids parent changes if no packet losses are observed despite the bandwidth performance being poor.

One of the parameters that we have found important is the *detection time* parameter, which indicates how long a host must stay with a poor performing parent before it switches to another parent. Our initial implementation employed a constant detection time of 5 seconds. However our experience reveals the need for the protocol to adaptively tune this timer because: (a) many hosts are not capable of receiving the full source rate, (b) even hosts that normally perform well may experience intermittent local network congestion, resulting in poor performance for any choice of parent, (c) there can be few good and available parent choices in the system. Changing parents under these environments may not be fruitful. We have implemented a simple heuristic for dynamically adjusting the detection time, involving an increase if several parent changes have been made recently, and a decrease if it has been a long time since the last parent change.

Parent Selection: When a host (say *A*) joins the broadcast, or needs to make a parent change, it probes a random subset of hosts it knows (30 in our implementation). The probing is biased toward members that have not been probed or have low delay. Each host *B* that responds to the probe provides information about: (i) the performance (application throughput in the recent 5 seconds, and delay) it is receiving; (ii) whether it is degree-saturated or not; and (iii) whether it is a descendant of *A* to prevent routing loops. The probe also enables *A* to determine the round-trip time to *B*. *A* waits for responses for 1 second, then eliminates those members that are saturated, or who are its descendant. It then evaluates the performance (throughput and delay) of the remaining hosts if it were to choose them as parents. If *A* does not have bandwidth estimates to potential parents, it picks one based on delay. Otherwise, it computes the expected application throughput as the minimum of the throughput *B* is currently seeing and the available bandwidth of the path between *B* and *A*. History of past performance is maintained so if *A* has previously chosen *B* as parent, then it has an estimate of the bandwidth of the overlay link *B* – *A*. *A* then evaluates how much improvement it could make if it were to choose *B*.

A switches to the parent *B* either if the estimated appli-

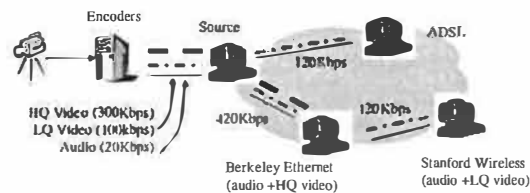


Figure 3: Single overlay approach to host heterogeneity.

cation throughput is high enough for *A* to receive a higher quality stream (see the multi-quality streaming discussion in § 2.3) or if *B* maintains the same bandwidth level as *A*'s current parent, but improves delay. This heuristic attempts to increase the tree efficiency by making hosts move closer to one another.

In order to assess the number of children a parent can support, we ask the user to choose whether or not it has at least a 10 Mbps up-link to the Internet. If so, we assign such hosts a degree bound of 6, to support up that many number of children. Otherwise, we assign a degree bound of 0 so that the host does not support any children. We have been experimenting with heuristics that can automatically detect the access bandwidth of the host, but this turns out not to be straightforward. We discuss this further in § 6.

2.2 Support for Receiver Heterogeneity

Internet hosts are highly heterogeneous in their receiving bandwidth, thus a *single-rate video coding* scheme is not the most appropriate. Various streaming systems have proposed using scalable coding techniques such layered coding or multiple description coding (MDC) in their design [35, 23, 5], however these technologies are not yet available in commercial media players. To strike a balance between the goals of rapid prototyping and heterogeneous receiver support, in our system, the source encodes the video at multiple bit-rates in parallel and broadcasts them simultaneously, along with the audio stream, through the overlay as shown in Figure 3. We run unicast congestion control on the data path between every parent and child, and a *prioritized packet forwarding* scheme is used to exploit the available bandwidth. That is, audio is prioritized over video streams, and lower quality video is prioritized over higher quality video. The system dynamically selects the best video stream based on loss rate to display to the user. Thus, audio is highly protected. When a receiver does not have sufficient bandwidth to view the high quality video stream, or when there are transient dips in available bandwidth due to congestions or poor parent choices, as long as the lower quality video stream is received, a legible image can still be displayed. We note that while this design involves some overhead, it can be seamlessly integrated with layered codecs if available.

Much of the deployment experience reported in this paper uses TCP as the congestion control protocol. We implement priority forwarding by having parents in the overlay tree maintain a fixed size per-child priority buffer. Packets are sent in strict priority and in FIFO order within each priority

class. If the priority buffer is full, packets are dropped in strict priority and in FIFO order (drop head). The priority buffer feeds the TCP socket, and we use *non-blocking write* for flow control. Note that once packets are queued in kernel TCP buffers, we can no longer control the prioritization. While we were aware of this limitation with using TCP, we were reluctant to employ untested UDP congestion control protocols in actual large scale deployment. Our subsequent experience has revealed that while the choice of TCP has only a minor hit on the performance of the prioritization heuristics, a more first-order issue is that it limits connectivity in the presence of NATs and firewalls. Faced with this, we have begun incorporating TFRC [12], a UDP-based congestion control protocol, into the system.

To prevent frequent quality switches that could annoy a user, we adopted a damping heuristic. Here, we aggressively switch to lower quality when high quality video has consistent loss for 10 seconds, and conservatively switch to higher quality when no loss is observed in the higher quality video stream for at least 50 seconds. Dynamically switching video qualities required us to implement an RTCP mixer[14]. When video qualities are switched, the mixer ensures the outgoing video stream to QuickTime is (i) masked as one contiguous stream; and (ii) time synchronized with the audio stream. One limitation in our current implementation is that if a host is displaying a low quality stream, the parent still forwards some data from the high quality stream. We are currently refining the implementation by adding heuristics to have the child unsubscribe from the higher quality stream, and periodically conduct experiments to see when network condition has improved so that it can start receiving the high quality stream.

2.3 Interface to Media Components

We use *QuickTime* [27] as the media player in our system because it is widely available and runs on multiple popular platforms. We use *Sorenson 3* [36] and MPEG4, both of which are supported by QuickTime, as video codecs. To support receiver heterogeneity, the source encodes the video at two target bit-rates (100 kbps and 300 kbps), and the audio at 20 kbps. We empirically determine the suitable encoding rates by experimenting with various encodings of conference talks. We find that a frame size of 640x480 is necessary to read the words on the slides. A minimal rate of 100 kbps yields watchable, 5 frames per second video motion. A rate of 300 kbps produces good video quality with 15 frames per second. To hide from the media player the fact that the overlay parent changes over time, we direct the player to a fixed *localhost:port* URL which points to the overlay proxy running at the same host. The overlay proxy handles all topology changes and sends data packets to the player as though it were a unicast streaming media server.

2.4 NATs and Firewalls

Our initial prototype did not include support for NATs and firewalls. We were motivated to address this as we consis-

Child	Parent		
	Public	NAT	Firewall
UDP Transport			
Public	✓	✓	✓
NAT	✓	?*	?
Firewall	✓	?	?*
TCP Transport			
Public	✓	✓	✓
NAT	✓	*	×
Firewall	✓	×	*

Table 1: Connectivity Matrix. ✓ means connectivity is always possible. ? means connectivity is possible for some cases of NAT/firewall and * means connectivity is only possible if the hosts are in the same private network.

tently needed to turn down 20 – 30% of viewers in our early broadcasts for the lack of such support. NATs and firewalls impose fundamental restrictions on pair-wise connectivity of hosts on the overlay. In most cases, it is not possible for NATs and firewalls to communicate directly with one another. However, there are specific exceptions, depending on the transport protocol (UDP or TCP), and the exact behavior of the NAT/firewall. Adopting the classification from STUN [15], *Full Cone NATs* can receive incoming packets to a port from any arbitrary host once it sends a packet on that port to any destination. Many hosts can address a host behind a full cone NAT using the same port number. In contrast, *Symmetric NATs* allow incoming packets only from the host that it has previously sent a packet to. Different hosts address a host behind a symmetric NAT using different port numbers. Table 1 characterizes these restrictions for the different transport protocols, where columns represent parents and rows represent children. For example, communication is not possible between two NATed hosts using TCP unless they happen to be in the same private network. In addition, “?” denotes that communication is possible using UDP between two NATed hosts if one of them is behind a *Full Cone NAT*. The *firewalls* which we refer to in Table 1 allow UDP packets to traverse in either direction. The system does not support firewalls that block UDP.

The primary goals in supporting NATs and firewalls are: (i) enable connectivity, a generic problem shared by many applications wishing to support these hosts and (ii) address protocol-specific enhancements to become “NAT/firewall-aware” to improve efficiency and performance.

2.4.1 Enable Connectivity

Use Overlay Identifier for Unique Naming: In the overlay protocol, each host needs to have a distinct and unique identifier. The straightforward use of public and private IP address and port does not serve this purpose because of symmetric NATs. To resolve this, we assign a unique overlay identifier(OID) to each host and decouple it from its IP address, separating overlay naming from addressing. When a host *A* joins the group, it is assigned an OID by the source. The source creates a binding that maps the OID of *A* to its public and private addresses and ports. This binding is distributed as part of the group membership management protocol.

Learn, Maintain, and Translate Bindings: There are two ways for a host *B* to learn bindings for host *A*. First, it can learn the binding as part of the group membership operations. Second, it may receive packets directly from *A*. Bindings learned by the second method are prioritized because they are the only ones that can be used to talk to a host behind a symmetric NAT. Each host *B* maintains the OID and associated binding for every other member *A* that it knows. The OID is translated into the appropriate binding when *B* wishes to send a packet to *A*. In some cases *A* and *B* may be behind the same private network, but have different public IP addresses. This is common in the case of large corporations that use multiple NAT gateways. We use a simple heuristic to match the prefixes in the public IP address. This matching expires if *B* does not receive packets from *A* after a short while.

Set up TCP Parent-Child Connection for Data: We use bi-directional connection initiation, by which both parent and child attempt to open a connection to the other. If one is a public and the other is NAT/firewall, then only one of the connections will be successful. If both are public, then both connections will be successful and we arbitrarily close the connection initiated by the host with higher IP address.

2.4.2 Making the Protocol Aware of NATs and Firewalls

The protocol works correctly with the connectivity service, without needing to make any changes. However, being aware of connectivity constraints can improve protocol efficiency and performance. We have identified 2 changes to the protocol to make it explicitly aware of connectivity constraints.

Group Management and Probing: To increase the efficiency of control messages, we enhance the group management protocol to explicitly avoid control messages between pairs of hosts that cannot communicate (e.g., NAT-NAT). Similarly, for probing, we do not allow NATs/firewalls to probe other NATs/firewalls.

Self-Organization: If the overlay protocol is aware of the NAT and firewall hosts in the system, it can support more of them by explicitly structuring the tree. For example, an efficient structure is one in which public hosts use NAT or firewall hosts as parents to the extent possible. In contrast, a structure in which a public host is a parent of another public host is inefficient because it reduces the potential parent resources for NAT hosts. While we have not deployed this mechanism, we evaluate its potential in § 6.

3 Deployment Status

3.1 System Status

To make the broadcast system easily and widely accessible, and attract as many participants as possible, we have taken effort to support multiple OS (Linux, Windows, MAC) and player platforms (QuickTime, Real Player) and develop user-friendly interfaces for both publishers and viewers. With the

subscriber Web interface, any receiver can tune in to a broadcast by a single click on a web-link.

The broadcast system is also designed for ease of deployment. We learned from our first broadcast event that having 5 graduate students spend 2 days to manually set up a broadcast was a barrier for deployment. Our publishing toolkit [11] has evolved since then into a user-friendly web based portal for broadcasting and viewing content. This portal allows content publishers to setup machines, machine profiles (such as which machines should be the source, log servers, and encoders), and events. With this information configured, the broadcast can be launched directly from the web. With no prior experience using the system and minimal support from us, most content publishers spend a couple hours to set up and run a broadcast. A monitoring system has been built to provide content publishers with online information about individual participating hosts, the current overlay tree, the bandwidth on each overlay link, and the current group membership. In addition, the system can recover from simple failures such as automatically re-starting the log server when it crashes.

As a research vehicle, the broadcast system has a built-in logging infrastructure that enables us to collect performance logs from all hosts participating in the broadcast for post-mortem analysis. The logs are sent on-line to a log server during the session. The data rate is bounded at 20 kbps to avoid interfering with the overlay traffic.

3.2 Deployment Experience

Over the last year, the system has been used by 4 content publishers and ourselves to broadcast more than 20 real events, the majority of which are conferences and lectures, accumulating 220 operational hours. In all, the system has been used by over 4000 participants. We summarize some of our key experience with regard to how successful we were in attracting publishers and viewers to use the system, the extent of our deployment, and some of the factors that affected our deployment.

Attracting content publishers: One of the key challenges we face is finding content. It has been difficult to access popular content such as movies and entertainment, as they are not freely available and often have copyright limitations. However, we have been more successful at attracting owners of technical content, such as conferences, workshops and lectures. Typically event organizers have expressed considerable interest in the use of our system. However given the wariness toward adopting new technology, convincing an event organizer to use the system involves significant time and groundwork. The key element of our success has been finding enthusiastic champions among conference organizers who could convince their more skeptical colleagues that it is worth their while to try the new technology even when they are already overwhelmed by all the other tasks that organizing a conference involves. We have also learned that the video production process is important, both in terms of cutting costs given that

conferences operate with low-budgets, and in terms of dealing with poor Internet connectivity from the conference sites to the outside world.

Viewer Participation: Table 2 lists the major broadcasts, duration, number of unique participants, and the peak group size. The broadcast events attracted from 15 to 1600 unique participants throughout the duration and peaked at about 10 to 280 simultaneous participants. Most of the audience tuned in because they were interested in the content, but could not attend the events in person. The Slashdot broadcast is different in that wanting to explore a larger scale and wider audience, we asked readers of Slashdot [34], a Web-based discussion forum, to experiment with our system. While some of the audience tuned in for the content, others tuned in because they were curious about the system.

While our deployment has been successful at attracting thousands of users, the peak group sizes in our broadcasts have been relatively low with the largest broadcast having a peak size of about 280. One possible explanation for this is that the technical content in these broadcasts fundamentally does not draw large peak group sizes. Another possibility is that users do not have sufficient interest in tuning in to live events, and prefer to view video archives. Our ongoing efforts to draw larger audience sizes include contacting non-technical organizations, and incorporating interactive features such as questions from the audience to the speaker.

We wish to emphasize that our limited operational experience with larger group sizes has been constrained by the lack of appropriate content, rather than due to specific known limitations of our system. We have had encouraging results evaluating our system in Emulab [40] using 1020 virtual nodes, multiplexed over 68 physical nodes, as well as simulation environments with over 10,000 nodes. Our hope is to use the workloads and traces of environment dynamics, resources and diversity from our broadcasts to design more realistic simulations and emulations in the future.

Diversity of Deployment: The diversity of hosts that took part in two of the large broadcasts (SIGCOMM 2002 and Slashdot), excluding waypoints, can be seen from Table 3. The deployment has reached a wide portion of the Internet - users across multiple continents, in home, academic and commercial environments, and behind various access technologies. We believe this demonstrates some of the enormous deployment potential of overlay multicast architectures - in contrast, the usage of the MBone [4] was primarily restricted to researchers in academic institutions.

Decoupling development version from deployment version: One of the challenges associated with operational deployment is the need for robust, well-tested and stable code. Bugs can not only affect the performance of a broadcast, but can also significantly lower our credibility with event organizers championing our cause. This requires us to adopt extensive testing procedures using Emulab [40], Planetlab [26], and Dummynet [31] before code is marked ready for deploy-

Event	Duration (hours)	Unique Hosts/ Waypoints	Peak Size/ Waypoints
SIGCOMM 2002	25	338/16	83/16
SIGCOMM 2003	72	705/61	101/61
DISC 2003	16	30/10	20/10
SOSP 2003	24	401/10	56/10
Slashdot	24	1609/29	160/19
DARPA Grand Challenge	4	800/15	280/15
Distinguished Lectures Series (8 distinct events)	9	358/139	80/59
Sporing Event	24	85/22	44/22
Commencement (3 distinct events)	5	21/3	8/3
Special Interest	14	43/3	14/3
Meeting	5	15/2	10/2

Table 2: Summary of major broadcasts using the system. The first 4 events are names of technical conferences.

ment. Further, in actual deployment, we typically use an older version of our system (several months) compared to our development version. One consequence of this is that even though certain design enhancements may seem trivial to incorporate, it may take several months before being used in actual broadcasts.

Use of Waypoints: Right from the early stages of our work on Overlay Multicast, we have been debating the architectural model for deploying Overlay Multicast. On the one hand, we have been excited by the deployment potential of *purely application end-point architectures* that do not involve any infrastructure support and rely entirely on hosts taking part in the broadcast. On the other hand, we have been concerned about the feasibility of these architectures, given that they depend on the ability of participating hosts to support other children. When it came to actual deployment, we were not in a position to risk the success of a real event (and consequently our credibility and the content provider's credibility) by betting on such an architecture. Thus, in addition to real participants, we employed PlanetLab [26] machines, which we call waypoints, to also join the broadcast (also listed in Table 2). From the perspective of the system, waypoints are the same as normal participating hosts and run the same protocol - the only purpose they served was increasing the amount of resources in the system. To see this, consider Figure 4, which plots a snapshot of the overlay during the *Conference* broadcast. The shape and color of each node represents the geographical location of the host as indicated by the legend. Nodes with a dark outer circle represent waypoints. There are two points to note. First, the tree achieves reasonable clustering, and nodes around the same geographical location are clustered together. Second, we see that waypoints are scattered around at interior nodes in the overlay, and may have used normal hosts as parents. Thus they behave like any other user, rather than statically provisioned infrastructure nodes. While our use of waypoints so far has prevented direct conclusions about purely application end-point architectures, we can arrive at important implications for these architectures leading to reduced use of waypoints in subsequent broadcasts, as we have done in § 6.

SIGCOMM 2002 broadcast 8/2002 9am-5pm (total 141 hosts)					
Region	North America (101)	Europe (20)	Oceania (1)	Asia (12)	Unknown (7)
Background	Home (26)	University (87)	Industry (5)	Government (9)	Unknown (14)
Connectivity	Cable Modem (12)	10+ Mbps (91)	DSL (14)	T1 (2)	Unknown (22)
Slashdot broadcast 12/2002 2pm-10:30pm (total 1316 hosts)					
Region	North America (967)	Europe (185)	Oceania (48)	Asia (8)	Unknown (108)
Background	Home (825)	University (127)	Industry (85)	Government (80)	Unknown (199)
Connectivity	Cable Modem (490)	10+ Mbps (258)	DSL (389)	T1 (46)	Unknown (133)
NAT	NAT (908)	Public (316)	Firewall (92)		

Table 3: Host distributions for two broadcast events, excluding waypoints, shown only for a portion of the broadcast.

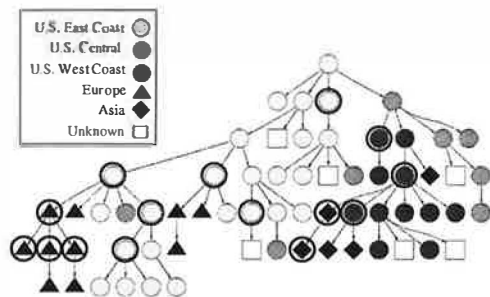


Figure 4: Snapshot of the overlay tree during Conference 1. Participants, marked by geographical regions, were fairly clustered. Waypoints, marked by outer circles, took on many positions throughout the tree.

4 Analysis Methodology

We conduct off-line analysis on the performance logs collected from hosts participating in the broadcasts. Our evaluation and analysis focus on the following questions:

- How well does the system perform in terms of giving good performance to the user?
- What kind of environments do we see in practice? How does the environment affect system performance? Are there quantitative indices we can use to capture environment information?
- Using trace-based simulations on the data, can we ask “what-if” questions and analyze design alternatives that could have led to better performance?

The data that we use for the analysis is obtained from performance logs collected from hosts participating in the broadcast. We have instrumented our system with measurement code that logs application throughput sampled at 1 second intervals, and application loss rate sampled at 5 second intervals. Note that the sample period is longer for loss rates because we found from experience that it is difficult to get robust loss measurements for shorter sampling periods.

We define an *entity* as a unique user identified by its $\langle \text{publicIP}, \text{privateIP} \rangle$ pair. An entity may join the broadcast many times, perhaps to tune in to distinct portions of the broadcast, and have many *incarnations*. The following sections, report analysis on incarnations unless otherwise stated.

Some of the analysis requires logs to be time synchronized. During the broadcast, whenever a host sends a message to the source as part of normal protocol operations (for example, gossip or probe message), the difference in local offsets is calculated and printed as part of the log. In the offline analysis, the global time for an event is reconstructed by adding

this offset. We have found that the inaccuracy of not considering clock skew is negligible.

In this section, we provide an overview of our analysis methodology. We present results from broadcasts in § 5. Finally, in § 6, we quantitatively analyze the performance benefits that may accrue from key design modifications motivated by our experience.

4.1 User Performance Metrics

We evaluate the performance that individual users observe by measuring their average and transient network-level performance. In addition, user-level feedback is also presented to provide a more complete picture of the user experience.

• **Average performance** is measured as the mean application-level throughput received at each incarnation. This provides a sense of the overall session performance.

• **Transient performance** is measured using the application-level losses that users experience. Using the sampled loss rate from the performance logs, we mark a sample as being a loss if its value is larger than 5% for each media stream, which in our experience is noticeable to human perception. We use three inter-related, but complementary metrics: (i) fraction of session for which the incarnation sees loss; (ii) mean interrupt duration; and (iii) interrupt frequency.

Fraction of session for which the incarnation sees loss is computed as follows. If an incarnation participates for 600 seconds, it would have about 120 loss samples. If 12 of those samples are marked as being a loss, then the incarnation sees loss for 10% of its session.

We define an interrupt to be a period of consecutive loss samples. Interrupt duration is computed as the amount of time that loss samples are consecutively marked as losses. The interrupt durations are then averaged across all interrupts that an incarnation experiences. Note that this metric is sensitive to the sampling period.

Interrupt frequency is computed as the number of distinct interrupts over the incarnation’s session duration, and reflects the dynamicity of the environment. A distinct interrupt is determined to be a consecutive period for which the loss samples are marked as a loss. This metric is biased by incarnations that have short session durations. For example, if an incarnation stays for 1 minute, and experiences 2 distinct 5-second interrupts, the interrupt frequency would be once every 30 seconds.

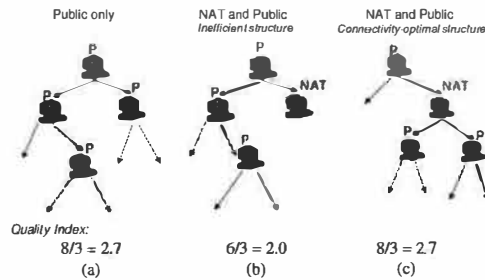


Figure 5: Example of Resource Index computation.

•**User Feedback** complements the network-level metrics described above. We encouraged users to fill in a feedback form and rate their satisfaction level for various quality metrics such as ease of setup, overall audio and video quality, frequency of stalls, and duration of stalls. The results are, however, subjective and should be considered in conjunction with the more objective network-level metrics.

•**Additional Metrics** to capture the quality of the overlay have also been analyzed. For example, we have looked at the efficiency of the overlay based on resource usage [9], and overlay stability based on the rate of parent changes. Due to space limitations, we do not present these results.

4.2 Environmental Factors

A self-organizing protocol needs to deal with events such as an ancestor leaving, or congestion on upstream overlay links by making parent changes. Two key factors that affect performance then are: (i) the dynamicity of the environment; and (ii) the availability of resources (parents) in the environment. The more dynamic an environment, the more frequently a host is triggered to react; the poorer the resources, the longer it could potentially take to discover a good parent.

4.2.1 Dynamics

The two key aspects of dynamics are: (i) group dynamics; and (ii) dynamics in the network. We measure group dynamics using mean interarrival time and session duration. We note however that the membership dynamics and overlay performance may not follow a strict cause and effect relationship. For example, users that see poor performance may leave, thus creating more dynamics in the system.

Our measurements are not conducive to summarizing network dynamics in terms of frequency and duration because of several reasons. First, we have measurements only for the subset of overlay links chosen and used by the protocol for data transfer. Second, the measurements could be biased by the protocol's behavior. For example, the observation of congestion duration may be shorter than in reality because the protocol attempts to move away from congestion and stops sampling that path. Instead, we characterize network dynamics by looking at the causes and location as described in § 4.3.

4.2.2 Environment Resources

Two key factors capture the resources in an environment: (i) outgoing bandwidth of hosts, which directly bounds the number of children hosts can take; and (ii) the presence of NATs and firewalls which places connectivity restrictions on parent-child relationships. In this section, we introduce a metric called the *Resource Index* to capture the outgoing bandwidth of hosts, and then extend it to consider NATs and firewalls.

We define the *Resource Index* as the ratio of the number of receivers that the members in the group could *potentially sustain* to the number of receivers in the group for a particular source rate. By number of hosts that can be potentially sustained, we mean the sum of the existing hosts in the system and the number of free slots in the system. For example, consider Figure 5(a), where each host has enough outgoing bandwidth to sustain 2 children. The number of free slots is 5, and the *Resource Index* is $(5 + 3)/3 = 8/3$. Further, for a given set of hosts and out-going bandwidth, the *Resource Index* is the same for any overlay tree constructed using these hosts. A *Resource Index* of 1 indicates that the system is saturated, and a ratio less than 1 indicates that not all the participating hosts in the broadcast can receive the full source rate. As the *Resource Index* gets higher, the environment becomes less constrained and it becomes more feasible to construct a good overlay tree. Note that the Resource Index is sensitive to the estimation of number of slots in the system.

We have extended the definition of *Resource Index* to incorporate the connectivity constraints of NATs and firewalls, by only considering free slots available for NAT hosts. For example, in Figure 5(b), the number of slots available for NAT hosts is 3, and the *Resource Index* is $6/3$. However, we note that the *Resource Index* not only depends on the set of hosts, but also becomes sensitive to the structure of the overlay for that set of hosts. Thus, while Figure 5(c) has the same set of hosts as Figure 5(b), we find the number of free slots for NATs is 5 and the *Resource Index* is $8/3$.

We observe that the optimal structure for accommodating NATs is one where public hosts preferentially choose NATs as parents, leaving more free slots at public hosts which NATs can then choose as parents. Based on this observation, the *optimal Resource Index* for a set of hosts involving NATs and firewalls is defined as S/N , where $S = S_{public} + \text{Min}(S_{nat}, N_{public})$. Here, S_{public} and S_{nat} are the maximum number of children that can be supported by the public and NAT hosts, N_{public} is the number of receivers that are public hosts and N is the total number of receivers. Figure 5(c) is an optimal structure for the set of hosts, and it can be verified that the formula confirms to the result stated above.

We wish to close with two practical issues that must be borne in mind with the *Resource Index*. First, it captures only the availability of resources in the environment, but does not account for factors such as performance of Internet paths. Also, the *Resource Index* is computed assuming global knowledge, but in practice, a distributed protocol may

not be able to use the resources as optimally as it could have.

4.3 Loss Diagnosis

When evaluating a self-organizing protocol, we need to distinguish between losses that could possibly be fixed by appropriate self-organization techniques from the losses that are fundamental to the system (i.e. those caused by access link capacity limitations, trans-oceanic bottleneck link congestions and local congestions). Further, we are interested in identifying the location of losses in the overlay tree, and attribute causes to the loss. We now summarize steps in our loss diagnosis methodology below:

- *Identifying Root-Events:* If a host sees bad performance, then all of its descendants downstream see bad performance. Our first step filters out losses at descendants, and isolates a set of “root-events”. If a host sees losses at a particular time, we determine whether its parent saw losses in a 5 second window around that time. This correlation relies on the time synchronization mechanism that we described earlier in the section.

- *Identifying Network Events:* Next, we classify the losses between the host and its parent based on cause. In our system, there are potentially two primary causes: (i) parent leave or death, and (ii) network problems (congestion or poor bandwidth) between the parent and child. There could be other miscellaneous causes such as host with slow processors and implementation bugs. Parent leave or death events are explicitly detected by the protocol and logged. Hosts with slow processors are detected by abnormal gaps in time-stamps of operations that log messages at periodic intervals. Implementation bugs are revealed by abnormal patterns we detect during manual verification and analysis of logs. Thus, after a detailed elimination process and exhaustive manual verification, we classify the remaining losses that we are not able to attribute to any known cause as due to network problems.

- *Classifying constrained hosts:* Network losses can occur at several locations: (i) local to the child where a parent change is not needed; or (ii) local to the parent, or on the link between parent and child. As a first step, we identify hosts that see persistent losses near it using the following heuristic. If a host has seen losses for over 80% of the session, all of which are “root losses”, and has tried at least 5 distinct parents during the session, then we decide the host is bandwidth constrained. Inherent here is the assumption that the protocol is doing a reasonable job in parent selection. This heuristic works well in environments with higher Resource Index. Finally, we manually verify these hosts and look for other evidence they are constrained (for example, location across a trans-oceanic link, names indicating they are behind wireless links etc.).

- *Classifying congestion losses:* The remaining losses correspond to hosts that usually see good performance but see transient periods of bad performance. If its siblings experience loss at around the same time, it is evidence that the loss is near the parent and not near a child; if a child has made

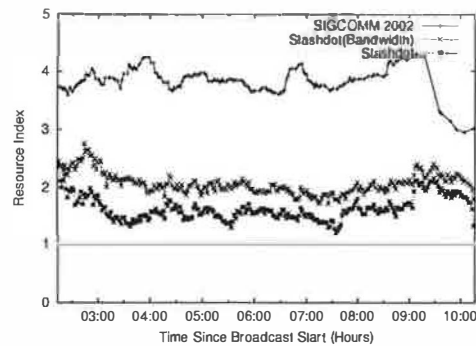


Figure 6: Resource Index as a function of time for (i) SIGCOMM 2002, (ii) Slashdot with bandwidth constraint, (iii) Slashdot with bandwidth and connectivity constraints.

several parent changes during an extended loss period, it is evidence that the loss is near the child. For the events that we are unable to classify, we label them as having “unknown location”.

5 Analysis Results

We present results from 6 of our larger broadcasts, 5 of which were conference/lecture-type broadcasts, and the other being *Slashdot*. For multi-day events, such as SIGCOMM 2002 and 2003, we analyzed logs from one day in the broadcast. For *Slashdot*, we present analysis results for the first 8 hours. In this section, we will present environment characterizations and performance results of the broadcasts. The analysis will indicate strong similarities in the environment for the conference/lecture-type broadcasts. However, they differ significantly from *Slashdot*. When we wish to illustrate a more detailed point, we use data from the *SIGCOMM 2002* and *Slashdot* broadcasts. The *SIGCOMM 2002* broadcast is one of the largest conference/lecture-type broadcasts, and is representative of these broadcasts in terms of application performance and resources.

5.1 Environment Dynamics

Table 4 lists the mean session interarrival time in seconds for the 6 broadcasts in the fourth column. For the five broadcasts of conferences and lectures, the mean interarrival time was a minute or more, whereas the interarrival time for *Slashdot* was just 17 seconds. *Slashdot* has the highest rate of group dynamics compared to all other broadcasts using our system. Note that the session interarrival times fit an exponential distribution.

Two different measures of session duration are listed in Table 4: individual incarnation duration and entity duration (cumulative over all incarnations) which captures the entity’s entire attention span. For entity session duration, again, we find that all 5 real broadcasts of conferences and lectures have a mean of 26 minutes or more, and a median of 16 minutes or more. In the *SIGCOMM 2002* broadcast, the median was 1.5 hours which corresponds to one technical session in the conference. To contrast, the *Slashdot* audience has a very short

Event	Duration (hours)	Incarnations Excluding Waypoints	Mean Session Interarrival Time (sec)	Incarnation Session Duration (minutes)		Entity Session Duration (minutes)		% Eligible Parents	
				Mean	Median	Mean	Median	NAT, Firewall, Public	Public
SIGCOMM 2002	8	375	83	61	11	161	93	57%	57%
SIGCOMM 2003	9	102	334	29	2	71	16	46%	17%
Lecture 1	1	52	75	12	2	26	19	62%	33%
Lecture 2	2	72	120	31	13	50	53	44%	21%
Lecture 3	1	42	145	31	7	42	31	73%	43%
Slashdot	8	2178	17	18	3	11	7	19%	7%

Table 4: Summary of group membership dynamics and composition for the 6 larger broadcasts using the system.

attention span of 11 and 7 minutes for the mean and median. This indicates that the Slashdot audience may have been less interested in the content. The incarnation session duration also follows a similar trend with shorter durations. Note that SIGCOMM 2003 and Lecture 1 have very short median incarnation session durations. This is caused by 1 or 2 entities testing the system, joining and leaving frequently. Once we removed such entities, the median went up to 12 minutes or more, bringing it closer to the other 3 conferences and lectures.

5.2 Environment Resources

We look at the percentage of incarnations in the system that were eligible as parents, the last 2 columns in Table 4. The 5 conference and lecture broadcasts have the same trend, with 44% or more incarnations that can serve as parents. On the other hand, only 19% of incarnations could be parents in Slashdot. Further, when we consider the fraction of *public* hosts that could be parents, we find this ranges from 17–57% for the conference-style broadcasts, but is just 7% for the Slashdot broadcast. This indicates that there were much less available resources in the system in the Slashdot broadcast. Note that we did not have NAT/firewall support in the SIGCOMM 2002 broadcast.

Figure 6 depicts the Resource Index of the system as a function of time of the broadcast. The top and the lowest curves represent the *Resource Index* for the SIGCOMM 2002 and Slashdot broadcasts, and are consistent with the definition in § 4.2.2. We note that the lowest curve corresponds to the actual overlay tree that was constructed during the broadcast. The middle curve, *Slashdot (Bandwidth)* considers a hypothetical scenario without connectivity constraints (that is, all NAT/firewall hosts are treated as public hosts). The SIGCOMM 2002 broadcast has a Resource Index of 4, potentially enough to support 4 times the number of members. In contrast, the *Slashdot (Bandwidth)* has a Resource Index of 2, and *Slashdot* has a Resource Index that is barely over 1. Thus, not only was the distribution of out-going bandwidth less favorable in the *Slashdot* broadcast, but also the presence of connectivity constraints made it a much harsher environment.

5.3 Performance Results

The previous analysis indicates that 5 of our broadcasts have similar resource distributions and dynamics patterns, but the

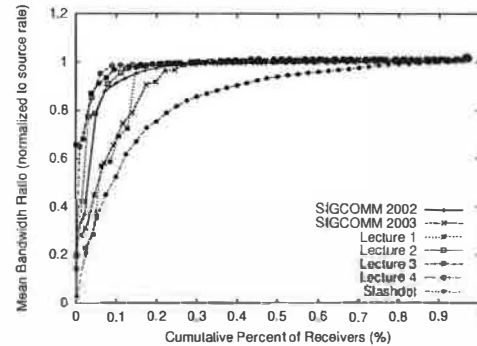


Figure 7: Cumulative distribution of mean session bandwidth (normalized to the source rate) for the 6 larger broadcasts.

	Setup ease	Audio Quality	Video Quality
SIGCOMM 2002	95%	92%	81%
Slashdot	96%	71%	66%

Table 5: Summary of user feedback for two broadcast events. Each number indicates the percentage of users who are satisfied in the given category.

Slashdot environment was more diverse and more dynamic. This section evaluates how the system performs.

Figure 7 plots the cumulative distribution of mean session bandwidth, normalized to the source rate for the 6 broadcasts. Five of the broadcasts are seeing good performance with more than 90% of hosts getting more than 90% of the full source rate in the SIGCOMM 2002, Lecture 2, and Lecture 3 broadcasts, and more than 80% of hosts getting more than 90% of the full source rate in the SIGCOMM 2003 and Lecture 1 broadcasts. In the Slashdot broadcast, fewer hosts, 60%, are getting the same performance of 90% of the full source rate.

To better understand the transient performance, and performance of different stream qualities, we zoom in on the SIGCOMM 2002, which we will refer to as *Conference*, and *Slashdot* broadcasts. Figure 8 depicts the cumulative distribution of the fraction of time all incarnations saw more than 5% packet losses in all three streams in Slashdot and the Conference broadcast, for incarnations that stay for at least 1 minute. For the Conference broadcast, the performance is good. Over 60% of the hosts see no loss in audio and low quality video, and over 40% of the hosts see no loss in high quality video. Further, over 90% of the hosts see loss for less than 5% of the session in the audio and low quality streams, and over 80% of the hosts see loss for less than 5% of the session in the high quality stream. We will further analyze

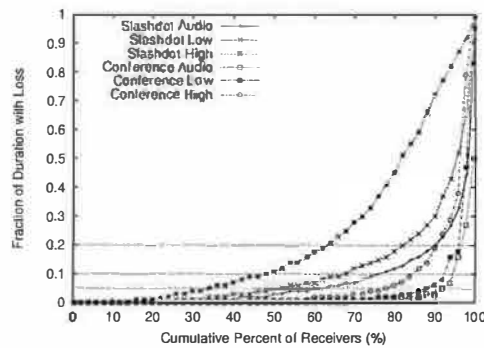


Figure 8: Cumulative distribution of fraction of session time with more than 5% packet loss of hosts in the two broadcasts.

the performance of the hosts that are seeing the worst performance in § 5.4 and demonstrate that these are mostly hosts that are fundamentally constrained by their access bandwidth. For the Slashdot broadcast on the other hand, the low quality video and audio streams see reasonable performance, but the performance of the high quality stream is much less satisfactory. Over 70% of the users see loss for less than 10% of the session in low quality video, but only 50% of users see loss for less than 10% of the session for high quality video. Note that the audio and low quality streams are seeing better performance than the high quality because of the use of the priority buffer described in § 2.2. For sessions with a high loss rate of high quality video, the low quality one was actually displayed to the user.

Next, we analyzed the interrupt duration and found that the interrupt duration is typically short for all 3 streams in Conference, and low quality video and audio in Slashdot. More than 70% of hosts see a mean interrupt duration of less than 10 seconds, and 90% of hosts see a mean interrupt duration of less than 25 seconds for all 5 streams. However, the high quality video in Slashdot sees a pronounced higher interrupt duration. Roughly 60% of hosts see a mean interrupt duration of longer than 10 seconds.

We have also analyzed the cumulative distribution of the frequency of interrupts seen by each incarnation. We find that the interrupt frequency is higher for Slashdot, probably reflecting the more dynamic environment. For example, in the Conference broadcast over 80% of hosts see an interrupt less frequent than once in five minutes and 90% see an interrupt less frequent than once in two minutes. In Slashdot, 60% of hosts see an interrupt less frequent than once in five minutes and 80% see an interrupt less frequent than once in two minutes.

User Feedback: Table 5 summarizes statistics from a feedback form users were encouraged to fill when they left the broadcast. Approximately 18% of users responded and provided feedback. Most users were satisfied with the overall performance of the system, and more satisfied with the overall performance in the Conference broadcast, which is consistent with the network level metrics in Figures 7 and 8.

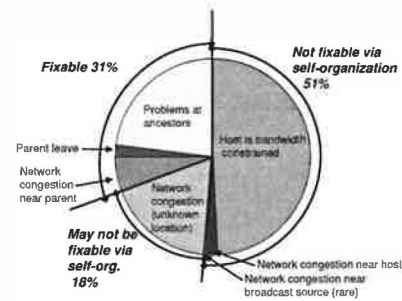


Figure 9: Loss diagnosis for Conference.

5.4 Loss Diagnosis

Figure 8 shows that for the *Conference* broadcast, while most users saw good performance, there is a tail which indicates poor performance. To better understand the tail, we analyze the data using the loss diagnosis methodology presented in § 4.3. Figure 9 shows the breakdown of all loss samples across all hosts. We find that almost 51% of losses are not fixable by self-organization. 49% corresponded to hosts that were bandwidth constrained, while 2% of losses belonged to hosts that were normally good, but experienced network problems close to them for a prolonged period. 6% of losses corresponded to network events that may be fixable by adaptation, while 18% of losses corresponded to network events that we were not able to classify. Manual cross-verification of the tail revealed about 30 incarnations that were marked as constrained hosts. This corresponded to about 17 distinct entities. Of these, 5 are in Asia, 1 in Europe, 3 behind wireless links, 1 behind a LAN that was known to have congestion issues, and 7 behind DSL links.

Finally, Figure 9 indicates that dynamics in the network is responsible for significantly more losses than group dynamics. In some cases, even well-provisioned paths see prolonged periods of congestion. As an anecdotal example, we observed that a gigabit link between a U.S. academic institution and the high-speed Internet2 backbone that typically provides good consistent performance, had a congestion epoch that lasted up to 3 minutes. Both observations are consistent with other broadcasts including Slashdot.

6 Lessons Learned

Our experience over the last year, substantiated with data and analysis, has pointed us toward four key design lessons that are guiding future refinements of our system.

Our first lesson sheds light on the potential of *purely application end-point based* overlay multicast architectures that rely entirely on the hosts taking part in the broadcast. As discussed in § 3.2, our deployment used waypoints, additional hosts that help increase the resources in the system but were otherwise no different than normal clients. We analyze how important the resources provided by waypoints was to the success of our broadcasts.

Our next three lessons deal with techniques that can enable good performance in environments with low Quality Index,

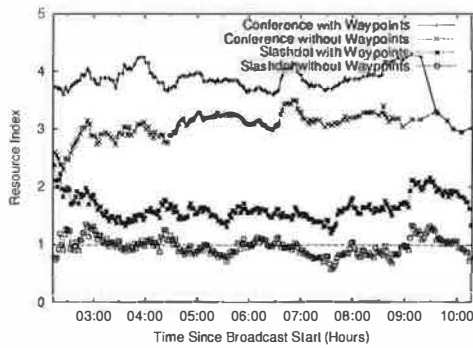


Figure 10: ResourceIndex as a function of time with and without waypoint support.

even in the absence of waypoints. The analysis for these lessons assume that the resources provided by waypoints is unavailable, and consequently a purely application end-point architecture.

Lesson 1: There is opportunity to reduce the dependence on waypoints and use them in an on-demand fashion.

In order to understand whether or not waypoints are necessary to the success of a broadcast, we look at Figure 10 which plots the *Resource Index* in the Conference and Slashdot broadcasts, with and without waypoints. The Conference broadcast had enough capacity to sustain all hosts even without waypoint support. Furthermore, most of the broadcasts, similar to the Conference broadcast, are sustainable using a purely application end-point architecture. In one of the lecture broadcasts, all the waypoint left simultaneously in the middle of the broadcast due to a configuration problem, and we found that the system was able to operate well without the waypoints.

On the other hand, we find that the connectivity constraints in the Slashdot broadcast resulted in a low *Resource Index* that occasionally dipped below 1 in Figure 10. This indicates that it was not feasible to construct an overlay among all participating hosts that could sustain the source rate. Dealing with such environments can take on two complementary approaches (i) design techniques that can enable good performance in purely application end-point architecture, even in the absence of waypoints (which forms the thrust of the subsequent lessons in this section), or (ii) use a waypoint architecture, with the insight that waypoints may not be needed for the entire duration of the broadcast, and can be invoked on-demand. For ease of deployment, our objective is to explore both approaches and gradually decrease the dependence on waypoints, using them as a back-up mechanism, only when needed.

We note that in the long-term, waypoint architectures may constitute an interesting research area in their own right, being intermediate forms between pure application end-point architectures and statically provisioned infrastructure-centric solutions. The key aspect that distinguishes waypoints

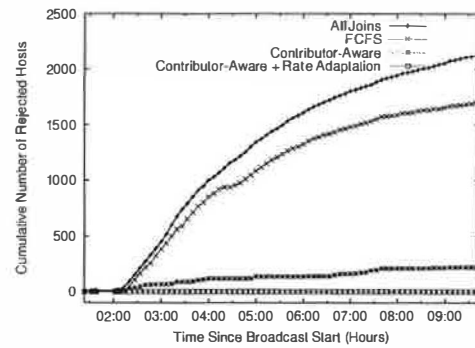


Figure 11: Number of rejected hosts under three different protocol scenarios in the simulated Slashdot environment.

from statically provisioned nodes is that the system does not depend on these hosts, but leverages them to improve performance.

Lesson 2: Exploiting heterogeneity in node capabilities through differential treatment is critical to improve the performance of the system in environments with low Resource Index. Further, there is considerable benefit to coupling such mechanisms with application-specific knowledge.

If the Resource Index dips below 1, the system must reject some hosts or degrade application quality. In this section, we evaluate performance in terms of the fraction of hosts that are rejected, or see lower application quality. We consider three policies. In the *First-Come-First-Served (FCFS)* policy that is currently used in our system, any host that is looking for a new parent, but finds no unsaturated parent is rejected. In the *Contributor-Aware* policy, the system distinguishes between two categories of hosts: contributors (hosts that can support children), and free-riders (hosts that cannot support children). A contributor C that is looking for a new parent may preempt a free-rider (say F). C can either accommodate F as a child, or kick it out of the system if C is itself saturated. This policy is motivated by the observation that preferentially retaining contributors over free-riders can help increase overall system resources. Finally, we consider *Rate-Adaptation* where a parent reduces the video rate to existing free-riders in order to accommodate more free-riders. For example, a parent can stop sending the high quality video (300 kbps) to one child, and in return, support three additional 100 kbps children. This policy is an example that not only differentially treats hosts based on their capabilities, but also exploits application knowledge.

We evaluate the potential of these policies by conducting a trace-based simulation using the group membership dynamics pattern from the Slashdot broadcast. We retain the same constitution of contributors and free-riders, but remove the waypoints from the group. We simulate a single-tree protocol where each receiver greedily selects an unsaturated parent, and we assume global knowledge in parent selection. If there is no unsaturated parent in the system, then we take action corresponding to the policies described above. Figure 11

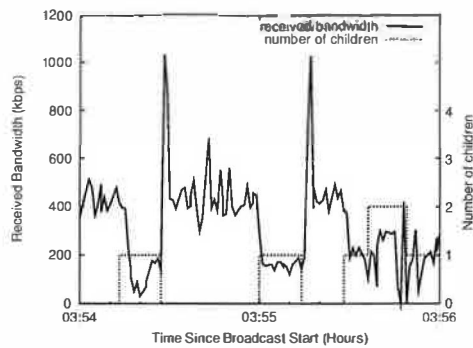


Figure 12: An example of a misconfigured DSL host taking children, causing poor performance to itself and its children.

	10+Mbps	Below 10Mbps	Total
User truthful	11.1%	60.8%	71.9%
User lied	5.4%	4.9%	10.3%
User inconsistent	4.3%	13.5%	17.8%
Total	20.8%	79.2%	100.0%

Table 6: Accuracy in determining access bandwidth based on user input in Slashdot.

shows the performance of the policies. We see that throughout the event, 78% of hosts are rejected using the *FCFS* policy. *Contributor-Aware* policy can drastically reduce the number of rejections to 11%. However, some free-riders are rejected because there are times when the system is saturated. With the *Rate Adaptation* policy however, no free-rider is rejected. Instead, 28% of the hosts get degraded video Resource for some portion of the session.

Our results demonstrate the theoretical potential of contributor-aware rejection and rate adaptation. A practical design has to deal with many issues, for example, robust ways of automatically identifying contributors (see next lesson), techniques to discover the saturation level of the system in a distributed fashion, and the trade-offs in terms of larger number of structure changes that preemption could incur. We are currently in the process of incorporating these policies in our design and evaluating their actual performance.

Lesson 3: Although many users are honest about contributing resources, techniques are needed for automatically estimating the outgoing access bandwidth of nodes.

As the previous lesson indicates, it is important to design protocol techniques that differentially treat nodes based on their contributions. An issue then is determining the contribution level of a node to the system, and in particular, determining the outgoing access bandwidth of a node. In our current system, the user is asked if his access bandwidth has a 10Mbps up-link to the Internet to help determine whether the host should have children (§ 2.1). This approach is susceptible to free-loaders[33], where a user declares that he has less resources than he really does. However, an equally damaging problem in the context of Overlay Multicast is when a user declares he has more resources than he does. To see this, consider Figure 12 which depicts the performance of a DSL

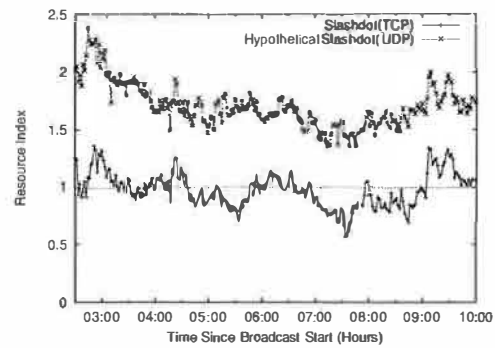


Figure 13: Resource Index comparison of two connectivity solutions for NAT/firewall: (i) Slashdot (TCP), (ii) Hypothetical Slashdot (UDP).

host that lied about having a 10Mbps up-link to the Internet, during the *Slashdot* broadcast. Whenever the host accepts a child, it affects not only the child's performance, but also its own performance. Further, a similar problem arises when a host can support less children (e.g. 4) than it claimed (e.g. 6). In a future design that prioritizes hosts that contribute more (Lesson 2), these effects can get further exacerbated.

To appreciate how reliable users were in selecting the correct access bandwidth in the *Slashdot* broadcast, consider Table 6. Each column represents a true access bandwidth, and each row represents a particular type of user behavior. "User Inconsistent" refers to users that had joined the group multiple times during the broadcast, and had selected both 10+Mbps option and lower than 10 Mbps option between consecutive joins, perhaps trying to figure out whether the choice yielded any difference in video quality. We determined the real access bandwidth using an off-line log analysis involving the following techniques: (i) DNS name, (ii) the TCP bandwidth of the upload log, (iii) online bottleneck bandwidth measurement, and (iv) Nettimer [18] from our university to target hosts. Since no single methodology is 100% accurate, we correlate results from all these techniques. We omit the details for lack of space.

From the table, we see that overall 71.9% of hosts are truthful. However, for the 20.8% of hosts that were behind 10Mbps links, only half of them (11.1% of total) were truthful. Our trace-based simulation on the *Slashdot* log indicates that on average, this results in a 20% increase in *Quality Index*. Further, we find that while 79.2% of the users were behind links lower than 10Mbps, about 4.9% chose the higher option or were being inconsistent (13.5%) about their connectivity.

We have been experimenting with techniques to explicitly measure the static outgoing access capacity of hosts and passively monitor the performance of parents to dynamically track their available bandwidth. These techniques show promise and we hope to deploy them in the future.

Lesson 4: Addressing the connectivity constraints posed by NATs and Firewalls may require using explicit NAT/firewall-aware heuristics in the protocol.

In light of our experience, NATs and firewalls can constitute an overwhelming fraction of a broadcast (for example, 50%-70% in *Slashdot*), and thus significantly lower the *Resource Index*. Clearly, using UDP as the transport protocol could improve the situation by increasing the amount of pair-wise connectivity, particularly connectivity between Full-Cone NATs. However, a less obvious improvement, which we briefly presented in § 2.4 is to make the self-organizing protocol explicitly aware of NAT/firewalls. In particular public hosts should preferentially choose NATs as parents, leaving more resources available for NATs/firewalls.

We now evaluate the potential of these two design improvements to help determine whether or not the additional complexity is worth the performance gains. Figure 13 shows the *Resource Index* for the system for the various design alternatives as a function of time, again omitting waypoint hosts. The lowest curve corresponds to the optimal *Resource Index* that can be achieved with a TCP-based protocol. The top-most curve corresponds to the optimal *Resource Index* with UDP and a NAT/firewall-aware self-organizing protocol. We see a significant increase of 74%. The combination of the two techniques above can significantly improve the *Resource Index*. Both techniques are being implemented in the latest version of our system and will soon be used for upcoming broadcasts.

7 Related Work

In this section, we discuss how our work relates to (i) other existing Internet broadcast systems and (ii) work in the Overlay Multicast community.

Broadcast Systems: The MBone [4] Project, and its associated applications such as vic [22], vat [16], and MASH [21] made a great effort to achieve ubiquitous Internet broadcasting. However, the MBone could only touch a small fraction of Internet users (mostly networking researchers) due to the fundamental limitations of IP Multicast and dependence on the special MBone infrastructure. In contrast, our system has over a short time already reached a wide range of users, including home users behind a range of access technologies, and users behind NATs and firewalls.

Commercial entities, such as Akamai [2] and Real Broadcast Network [29], already provide Internet broadcasting as a charged service. They rely on dedicated, well-provisioned infrastructure nodes to replicate video streams. Such an approach has some fundamental advantages such as security and stable performance. However, these systems are viable only for larger-scale publishers, rather than the wide-range of low budget Internet broadcasting applications we seek to enable.

Recently, several peer-to-peer broadcast systems have been built by commercial entities [3, 6, 38] and non-profit organizations [24]. To our knowledge, many of these systems focus on audio applications which have lower bandwidth requirements. However, given the limited information on these systems, we are unable to do a detailed comparison.

Overlay Multicast: Since overlay multicast was first proposed four years ago many efforts [13, 9, 17, 7, 19, 28, 37, 20, 32, 23, 39, 10, 5] have advanced our knowledge on protocol construction by improving performance and scalability. Most of this work has been *protocol-centric*, and has primarily involved evaluation in simulation, and Internet testbeds such as PlanetLab. In contrast, this paper adopts an *application-centric* approach, which leverages experience from actual deployment to guide the research. We address a wide range of issues such as support for heterogeneous receivers, and NATs and firewalls, which are not typically considered in protocol design studies. To our knowledge this paper is among the first reports on experience with a real application deployment based on overlay multicast involving real users watching live content. We believe our efforts complement ongoing research in overlay multicast, by validation through real deployment, and providing unique data, traces and insight that can guide future research.

The overlay protocol that we use is distributed, self-organizing and performance-aware. We use a distributed protocol, as opposed to a centralized protocol [25, 23], to minimize the overhead at the source. The self-organizing protocol constructs an overlay tree amongst participating hosts in a tree-first manner, similar to other protocols [17, 39, 13], motivated by the needs of single source applications. In contrast there are protocols that construct a richer mesh structure first and then construct a tree on top [9, 7], or construct DHT-based meshes using logical IDs and employ a routing algorithm to construct a tree in the second phase [20]. Such protocols are typically designed for multi-source or multi-group applications.

In our protocol, members maintain information about hosts that may be uncorrelated to the tree, in addition to path information, while in protocols like Overcast [17] and NICE [32], group membership state is tightly coupled to the existing tree structure: While Yoid [13] and Scribe [20] also maintain such information, the mechanisms they adopt are different. Our system uses a gossip protocol adapted from [30], while Yoid builds a separate random control structure called the mesh, and Scribe constructs a topology based on logical identifiers.

Overcast [17] and Narada [9] discuss adaptation to dynamic network metrics such as bandwidth. Our experience indicates that a practical deployment must consider several details such as dynamic tuning of network detection time to the resources available in the environment, consider hosts that cannot sustain the source rate, and consider VBR streams, and indicate the need for further research and understanding in this area.

Recent work such as CoopNet [23], and Splitstream [5] has demonstrated significant benefits by tightly coupling codec-specific knowledge and overlay design. In these works, the source uses a custom codec to encode the multimedia stream into many sub-streams using multiple description coding, and constructs an overlay tree to distribute each sub-

stream. This approach not only increases overall resiliency of the system, but also enables support for heterogeneous hosts by having each receiver subscribe to as many layers as its capacity allows. While we believe this a great direction for future research, our design has been influenced by practical system constraints on an immediately deployable operational system, and our desire to interoperate with commercial media players and a wide range of popular codecs. We hope to leverage ideas from this approach as the research attains greater maturity, and when custom codecs become available.

NATs and Firewalls: Several efforts such as UPnP [1] and STUN [15] focus their efforts in enabling connectivity of NATs and firewalls. Our focus in this paper has been on the interplay between the application and NAT/firewall support. In particular, we have examined how the connectivity constraints imposed by NATs and firewalls can impact overlay performance, and on issues related to the integration of protocol design with NATs and firewalls. While Yoid [13] supports NATs and firewalls, it supports such hosts as children only, whereas we try to use NATs as parents when possible. We believe this is one of the first reports on experience with an overlay multicast system in the presence of NATs and firewalls.

8 Summary and Future Work

In this paper, we have reported on our operational experience with a broadcast system based on Overlay Multicast. To our knowledge this is among the first reports on experience with real application deployment based on Overlay Multicast, involving real users. Our experience has included several positives, and taught us important lessons both from an operational deployment stand-point, and from a design stand-point.

Our system is satisfying the needs of real content publishers and viewers, and demonstrating the potential of Overlay Multicast as a cost-effective alternative for enabling Internet broadcast. The system is easy to use for both publishers and viewers. We have successfully attracted over 4000 users from diverse Internet locations to use our system. However, we have had limited success in attracting larger scales of participation, primarily because of the difficulty in getting access to non-technical content. Our experience with several conference/lecture-type broadcasts indicate that our system provides good performance to users. In such environments, we consistently observe that over 80 – 90% of the hosts see loss for less than 5% of their sessions. Further, hosts that perform poorly are typically bandwidth constrained hosts. Even in a more extreme environment with a low *Resource Index*, users see good performance in audio and low Resource video.

Getting the system deployed has frequently required finding an enthusiastic champion of the technology to convince their colleagues to use it. This has raised the stakes to ensure the success of a broadcast, which could in turn trigger further interest in the use of the system. Consequently, we have needed to use stable and well-tested code in our deployment,

rather than code that implements the latest performance enhancements. Another consequence has been our use of waypoints, additional hosts that help increase the resources in the system, but were otherwise no different than normal clients. The use of waypoints has been motivated by the need to balance between conflicting goals - on the one hand we want to understand the resource availability in purely application end-point architectures; on the other hand we need to have a series of successful broadcasts in the first place before such knowledge can be obtained.

Our subsequent analysis has investigated the potential of *purely application end-point architectures*, that do not rely on the use of waypoints. Our analysis both show the promise for such architectures, but also the need to incorporate additional key design elements. For most of our broadcasts, there is sufficient bandwidth resources to enable a solution purely within the application end-point framework. In broadcasts with lower Resource Index, techniques that exploit the heterogeneity in node capabilities through differential treatment and application-specific knowledge bear significant promise. Our broadcasts have also forced us to better appreciate the connectivity constraints posed by NATs and firewalls, and have led us to investigate explicit NAT/firewall-aware heuristics in the protocol. While our lessons have been derived in the context of our system, we believe they are of broader applicability to the community as a whole.

With the experience accumulated over the last year, we have set several milestones for the next 1 year horizon. Our milestones include:

- At a design level, we hope to incorporate some of the design refinements described above which can enable better performance in purely application end-point architectures. Our hope is to gradually minimize dependence on waypoints, through the use of on-demand waypoint invocation mechanisms.
- At an operational level, we hope to pursue wider and larger-scale deployment by attracting more publishers of both technical and non-technical content to the system, and convincing them to conduct their own broadcasts, incorporating interactivity features that might attract larger scales in synchronous applications, and encouraging other groups to run the broadcasts. Finally, while we have been conducting studies on the scalability of the system using emulations and simulations, we hope to gain deployment experience with larger peak group sizes.

In this paper, we use the broadcast system as a research vehicle for networking. However, we believe it has value extending into other areas of research. For example, one potential direction is in social science: is there a demand for interactivity in video broadcast and what are effective means of interactivity? Unlike traditional TV broadcast, Internet broadcast provides opportunities for viewers to actively engage in the event. We believe interaction among viewers are new and valuable social capital that did not previously exist in

traditional TV broadcast. If interactivity is used effectively, it can enhance the viewing experience, create positive feedback, and grow into a virtual community.

Availability

Our system is available at <http://esm.cs.cmu.edu>.

Acknowledgements

This research was sponsored by DARPA under contract number F30602-99-1-0518, by NSF under grant numbers Career Award NCR-9624979 ANI-9730105, ITR Award ANI-0085920, and ANI-9814929, and by the Texas Advanced Research Program under grant No. 003604-0078-2003. Additional support was provided by Intel. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, Texas ARP, Intel, or the U.S. government. Special thanks go to James Crugnale, Brian Goodman, Tian Lin, Nancy Miller, Jiin Joo Ong, Chris Palow, Vishal Soni, and Philip Yam for the help with the implementation and experimentation of the broadcast system. We also thank the early adopter event organizers who use our system to broadcast their events over the Internet, and the early adopter users who use our system to view the content. Finally, we thank our anonymous reviewers for their feedback and insight.

References

- [1] Understanding Universal Plug and Play. Microsoft White Paper.
- [2] Akamai. <http://www.akamai.com/>.
- [3] Allcast. <http://www.allcast.com/>.
- [4] S. Casner and S. Deering. First IETF Internet audiocast. *ACM Computer Communication Review*, pages 92–97, 1992.
- [5] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth Content Distribution in Cooperative Environments. In *Proceedings of SOSP*, 2003.
- [6] Chaincast. <http://www.chaincast.com/>.
- [7] Y. Chawathe. Scattercast: An architecture for Internet broadcast distribution as an infrastructure service. Fall 2000. Ph.D. thesis, U.C. Berkeley.
- [8] Y. Chu, S.G. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, August 2001.
- [9] Y. Chu, S.G. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, June 2000.
- [10] J. Albrecht D. Kotic, A. Rodriguez and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of SOSP*, 2003.
- [11] End system multicast toolkit and portal. <http://esm.cs.cmu.edu/>.
- [12] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Multicast Routing in Internetworks and Extended LANs. In *Proceedings of the ACM SIGCOMM*, August 2000.
- [13] P. Francis. Yoid: Your Own Internet Distribution, <http://www.aciri.org/yoid/>. April 2000.
- [14] R. Frederick H. Schulzrinne, S. Casner and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC-1889, January 1996.
- [15] C. Huitema J. Rosenberg, J. Weinberger and R. Mahy. STUN - Simple Traversal of UDP Through Network Address Translators. IERF-Draft, December 2002.
- [16] V. Jacobson and S. McCanne. Visual Audio Tool (vat). In *Audio Tool (vat)*, Lawrence Berkley Laboratory. Software online, <ftp://ftp.ee.lbl.gov/conferencing/vat>.
- [17] J. Jannotti, D. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.
- [18] K. Lai and M. Baker. Nettimer: A Tool for Measuring Bottleneck Link Bandwidth. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2001.
- [19] J. Liebeherr and M. Nahas. Application-layer Multicast with Delaunay Triangulations. In *IEEE Globecom*, November 2001.
- [20] A.M. Kermarrec M. Castro, P. Druschel and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. In *IEEE Journal on Selected Areas in Communications Vol. 20 No. 8*, Oct 2002.
- [21] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T. L. Tung, D. Wu, and B. Smith. Toward a Common Infrastructure for Multimedia-Networking Middleware. In *Proceedings of NOSSDAV*, 1997.
- [22] S. McCanne and V. Jacobson. vic: A Flexible Framework for Packet Video. In *ACM Multimedia*, November 1995.
- [23] V.N. Padmanabhan, H.J. Wang, and P.A Chou. Resilient Peer-to-peer Streaming. In *Proceedings of IEEE ICNP*, 2003.
- [24] Peercast. <http://www.peercast.org/>.
- [25] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An Application Level Multicast Infrastructure. In *Proceedings of 3rd Usenix Symposium on Internet Technologies & Systems (USITS)*, March 2001.
- [26] Planetlab. <http://www.planet-lab.org/>.
- [27] Quicktime. <http://www.apple.com/quicktime>.
- [28] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Proceedings of NGC*, 2001.
- [29] Real broadcast network. <http://www.real.com/>.
- [30] R. Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Cornell University Computer Science, 1998.
- [31] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. In *ACM Computer Communication Review*, January 1997.
- [32] B. Bhattacharjee S. Banerjee and C. Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, August 2002.
- [33] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, January 2002.
- [34] Slashdot. <http://www.slashdot.org/>.
- [35] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of ACM SIGCOMM*, August 1996.
- [36] Sorenson. <http://www.sorenson.com/>.
- [37] S.Q. Zhuang, B.Y. Zhao, J.D. Kubiawicz, and A.D. Joseph. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination, April 2001. Unpublished Report.
- [38] Streamer. <http://streamerp2p.com/>.
- [39] W. Wang, D. Helder, S. Jamin, and L. Zhang. Overlay optimizations for end-host multicast. In *Proceedings of Fourth International Workshop on Networked Group Communication (NGC)*, October 2002.
- [40] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI02*, pages 255–270, Boston, MA, December 2002.

Reliability and Security in the CoDeeN Content Distribution Network

Limin Wang*, KyoungSoo Park, Ruoming Pang, Vivek Pai and Larry Peterson
Department of Computer Science
Princeton University

Abstract

With the advent of large-scale, wide-area networking testbeds, researchers can deploy long-running distributed services that interact with other resources on the Web. The CoDeeN Content Distribution Network, deployed on PlanetLab, uses a network of caching Web proxy servers to intelligently distribute and cache requests from a potentially large client population. We have been running this system nearly continuously since June 2003, allowing open access from any client in the world. In that time, it has become the most heavily-used long-running service on PlanetLab, handling over four million accesses per day. In this paper, we discuss the design of our system, focusing on the reliability and security mechanisms that have kept the service in operation.

Our reliability mechanisms assess node health, preventing failing nodes from disrupting the operation of the overall system. Our security mechanisms protect nodes from being exploited and from being implicated in malicious activities, problems that commonly plague other open proxies. We believe that future services, especially peer-to-peer systems, will require similar mechanisms as more services are deployed on non-dedicated distributed systems, and as their interaction with existing protocols and systems increases. Our experiences with CoDeeN and our data on its availability should serve as an important starting point for designers of future systems.

1 Introduction

The recent development of Internet-scale network testbeds, such as PlanetLab, enables researchers to develop and deploy large-scale, wide-area network projects subjected to real traffic conditions. Previously, such systems have either been commercial enterprises (e.g., content distribution networks, or CDNs), or have been community-focused distributed projects (e.g., free file-sharing networks). If we define a design space of latency versus throughput and tightly-controlled versus decentralized management, we can see that existing CDNs and file-sharing services occupy three portions of the

space. The remaining portion, latency-sensitive decentralized systems, remains more elusive, without an easily-identifiable representative. In this paper, we describe CoDeeN, an academic Content Distribution Network deployed on PlanetLab, that uses a decentralized design to address a latency-sensitive problem.

To reduce access latency, content distribution networks use geographically distributed server surrogates, which cache content from the origin servers, and request redirectors, which send client requests to the surrogates. Commercial CDNs [2, 23] replicate pages from content providers and direct clients to the surrogates via custom DNS servers often coupled with URL rewriting by the content providers. The infrastructure for these systems is usually reverse-mode proxy caches with custom logic that interprets rewritten URLs. This approach is transparent to the end user, since content providers make the necessary changes to utilize the reverse proxies.

Our academic testbed CDN, CoDeeN, also uses caching proxy servers, but due to its non-commercial nature, engages clients instead of content providers. Clients must currently specify a CoDeeN proxy in their browser settings, which makes the system demand-driven, and allows us to capture more information on client access behavior. Given the high degree of infrastructural overlap, our future work may include support for non-commercial content providers, or even allowing PlanetLab members to automatically send their HTTP traffic to CoDeeN by using transparent proxying.

As shown in Figure 1, a CoDeeN instance consists of a proxy operating in both forward and reverse modes, as well as the redirection logic and monitoring infrastructure. When a client sends requests to a CoDeeN proxy, the node acts as a forward proxy and tries to satisfy the requests locally. Cache misses are handled by the redirector to determine where the request should be sent, which is generally another CoDeeN node acting as the reverse proxy for the origin server. For most requests, the redirector considers request locality, system load, reliability, and proximity when selecting another CoDeeN node. The reliability and security mechanisms can exclude nodes from being candidates, and can also reject requests entirely for various reasons described later.

* current contact: Dept of EECS, Case Western Reserve University

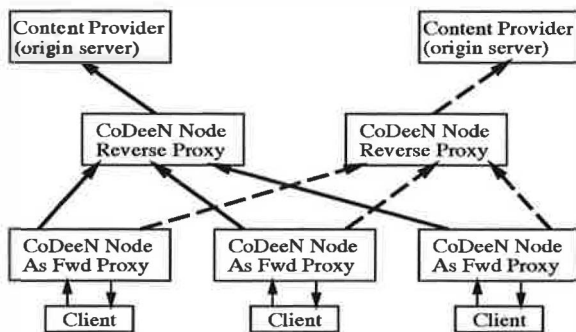


Figure 1: CoDeeN architecture – Clients configure their browsers to use a CoDeeN node, which acts as a forward-mode proxy. Cache misses are deterministically hashed and redirected to another CoDeeN proxy, which acts as a reverse-mode proxy, concentrating requests for a particular URL. In this way, fewer requests are forwarded to the origin site.

Although some previous research has simulated caching in decentralized/peer-to-peer systems [13, 26], we believe that CoDeeN is the first deployed system, and one key insight in this endeavor has been the observation that practical reliability is more difficult to capture than traditional fail-stop models assume. In our experience, running CoDeeN on a small number of PlanetLab nodes was simple, but overall system reliability degraded significantly as nodes were added. CoDeeN now runs on over 100 nodes, and we have found that the status of these proxy nodes are much more dynamic and unpredictable than we had originally expected. Even accounting for the expected problems, such as network disconnections and bandwidth contention, did not improve the situation. In many cases, we found CoDeeN unsuccessfully competing with other PlanetLab projects for system resources, leading to undesirable behavior.

The other challenging aspect of CoDeeN’s design, from a management standpoint, is the decision to allow all nodes to act as “open” proxies, accepting requests from any client in the world instead of just those at organizations hosting PlanetLab nodes. This decision makes the system more useful and increases the amount of traffic we receive, but the possibility of abuse also increases the chances that CoDeeN becomes unavailable due to nodes being disconnected. However, we overestimated how long it would take for others to discover our system and underestimated the scope of activities for which people seek open proxies. Within days of CoDeeN becoming stable enough to stay continuously running, the PlanetLab administrators began receiving complaints regarding spam, theft of service, abetting identity theft, etc.

After fixing the discovered security-related problems, CoDeeN has been running nearly continuously since June 2003. In that time, it has received over 300 million requests from over 500,000 unique IP addresses (as of December 2003), while generating only three com-

plaints. Node failure and overload are automatically detected and the monitoring routines provide useful information regarding both CoDeeN and PlanetLab. We believe our techniques have broader application, ranging from peer-to-peer systems to general-purpose monitoring services. Obvious beneficiaries include people deploying open proxies for some form of public good, such as sharing/tolerating load spikes, avoiding censorship, or providing community caching. Since ISPs generally employ transparent proxies, our techniques would allow them to identify customers abusing other systems before receiving complaints from the victims. We believe that any distributed system, especially those that are latency-sensitive or that run on non-dedicated environments, can benefit from our infrastructure for monitoring and avoidance.

The rest of the paper is organized as follows. In Section 2, we discuss system reliability and CoDeeN’s monitoring facilities. We discuss the security problems facing CoDeeN in Section 3, followed by our remedies in Section 4. We then show some preliminary findings based on the data we collected and discuss the related work.

2 Reliability and Monitoring

Unlike commercial CDNs, CoDeeN does not operate on dedicated nodes with reliable resources, nor does it employ a centralized Network Operations Center (NOC) to collect and distribute status information. CoDeeN runs on all academic PlanetLab sites in North America, and, as a result, shares resources with other experiments.¹ Such sharing can lead to resource exhaustion (disk space, global file table entries, physical memory) as well as contention (network bandwidth, CPU cycles). In such cases, a CoDeeN instance may be unable to service requests, which would normally lead to overall service degradation or failure. Therefore, to maintain reliable and smooth operations on CoDeeN, each instance monitors system health and provides this data to its local request redirector.

In a latency-sensitive environment such as CoDeeN, avoiding problematic nodes, even if they (eventually) produce a correct result, is preferable to incurring reliability-induced delays. Even a seemingly harmless activity such as a TCP SYN retransmit increases user-perceived latency, reducing the system’s overall utility. For CoDeeN to operate smoothly, our distributed redirectors need to continually know the state of other proxies and decide which reverse proxies should be used for request redirection. In practice, what this entails is first finding a healthy subset of the proxies and then letting the redirection strategy decide which one is the best. As a result, CoDeeN includes significant node health monitoring facilities, much of which is not specific to CoDeeN and can be used in other latency-sensitive peer-to-peer environments.

¹Resource protection in future PlanetLab kernels will mitigate some problems, but this feature may not exist on non-PlanetLab systems.

Two alternatives to active monitoring and avoidance, using retry/failover or multiple simultaneous requests, are not appropriate for this environment. Retrying failed requests requires that failure has already occurred, which implies latency before the retry. We have observed failures where the outbound connection from the reverse proxy makes no progress. In this situation, the forward proxy has no information on whether the request has been sent to the origin server. The problem in this scenario is the same reason why multiple simultaneous requests are not used – the idempotency of an HTTP request can not be determined *a priori*. Some requests, such as queries with a question mark in the URL, are generally assumed to be non-idempotent and uncacheable. However, the CGI mechanism also allows the query portion of the request to be concatenated to the URL as any other URL component. For example, the URL `"/directory/program/query"` may also be represented as `"/directory/program?query"`. As a result, sending multiple parallel requests and waiting for the fastest answer can cause errors.

The success of distributed monitoring and its effectiveness in avoiding problems depends on the relative difference in time between service failures and monitoring frequency. Our measurements indicate that most failures in CoDeeN are much longer than the monitoring frequency, and that short failures, while numerous, can be avoided by maintaining a recent history of peer nodes. The research challenge here is to devise effective distributed monitoring facilities that help to avoid service disruption and improve system response latency. Our design uses heartbeat messages combined with other tests to estimate which other nodes are healthy and therefore worth using.

2.1 Local Monitoring

Local monitoring gathers information about the CoDeeN instance's state and its host environment, to assess resource contention as well as external service availability. Resource contention arises from competition from other processes on a node, as well as incomplete resource isolation. External services, such as DNS, can become unavailable for reasons not related to PlanetLab.

We believe that the monitoring mechanisms we employ on PlanetLab may be useful in other contexts, particularly for home users joining large peer-to-peer projects. Most PlanetLab nodes tend to host a small number of active experiments/projects at any given time. PlanetLab uses *vservers*, which provide a view-isolated environment with a private root filesystem and security context, but no other resource isolation. While this system falls short of true virtual machines, it is better than what can be expected on other non-dedicated systems, such as multi-tasking home systems. External factors may also be involved in affecting service health. For example, a site's DNS server failure can disrupt the CoDeeN instance, and most of these problems appear to be external to PlanetLab [17].

The local monitor examines the service's primary resources, such as free file descriptors/sockets, CPU cycles, and DNS resolver service. Non-critical information includes system load averages, node and proxy uptimes, traffic rates (classified by origin and request type), and free disk space. Some failure modes were determined by experience – when other experiments consumed all available sockets, not only could the local node not tell that others were unable to contact it, but incoming requests appeared to be indefinitely queued inside the kernel, rather than reporting failure to the requester.

Values available from the operating system/utilities include node uptime, system load averages (both via `/proc`), and system CPU usage (via `vmstat`). Uptime is read at startup and updated inside CoDeeN, while load averages are read every 30 seconds. Processor time spent inside the OS is queried every 30 seconds, and the 3-minute maximum is kept. Using the maximum over 3 minutes reduces fluctuations, and, at 100 nodes, exceeds the gap between successive heartbeats (described below) from any other node. We avoid any node reporting more than 95% system CPU time, since we have found it correlates with kernel/scheduler problems. While some applications do spend much time in the OS, few spend more than 90%, and 95% generally seems failure-induced.

Other values, such as free descriptors and DNS resolver performance, are obtained via simple tests. We create and destroy 50 unconnected sockets every 2 seconds to test the availability of space in the global file table. At our current traffic levels, 50 sockets are generally sufficient to handle two seconds of service on a single node. Any failures over the past 32 attempts are reported, which causes peers to throttle traffic for roughly one minute to any node likely to fail. Similarly, a separate program periodically calls `gethostbyname()` to exercise the node's DNS resolver. To measure comparable values across nodes, and to reduce off-site lookup traffic, only other (cacheable) PlanetLab node names are queried. Lookups requiring more than 5 seconds are deemed failed, since resolvers default to retrying at 5 seconds. We have observed DNS failures caused by misconfigured `"/etc/resolv.conf"` files, periodic heavyweight processes running on the name servers, and heavy DNS traffic from other sources.

2.2 Peer Monitoring

To monitor the health and status of its peers, each CoDeeN instance employs two mechanisms – a lightweight UDP-based heartbeat and a “heavier” HTTP/TCP-level “fetch” helper. These mechanisms are described below.

2.2.1 UDP Heartbeat

As part of its tests to avoid unhealthy peers, CoDeeN uses UDP heartbeats as a simple gauge of liveness. UDP has low overhead and can be used when socket exhaustion prevents TCP-based communication. Since it is unreliable, only small amounts of non-critical information are

sent using it, and failure to receive acknowledgements (ACKs) is used to infer packet loss.

Each proxy sends a heartbeat message once per second to one of its peers, which then responds with information about its local state. The piggybacked load information includes the peer's average load, system time CPU, file descriptor availability, proxy and node uptimes, average hourly traffic, and DNS timing/failure statistics. Even at our current size of over 100 nodes, this heartbeat traffic is acceptably small. For larger deployments, we can reduce heartbeat frequency, or we may divide the proxies into smaller groups that only exchange aggregate information across groups.

Heartbeat acknowledgments can get delayed or lost, giving some insight into the current network/node state. We consider acknowledgments received within 3 seconds to be acceptable, while any arriving beyond that are considered "late". The typical inter-node RTT on CoDeeN is less than 100ms, so not receiving an ACK in 3 seconds is abnormal. We maintain information about these late ACKs to distinguish between overloaded peers/links and failed peers/links, for which ACKs are never received.

Several policies determine when missing ACKs are deemed problematic. Any node that does not respond to the most recent ACK is avoided, since it may have just recently died. Using a 5% loss rate as a limit, and understanding the short-term nature of network congestion, we avoid any node missing 2 or more ACKs in the past 32, since that implies a 6% loss rate. However, we consider viable any node that responds to the most recent 12 ACKs, since it has roughly a 54% chance of having 12 consecutive successes with a 5% packet loss rate, and the node is likely to be usable.

By coupling the history of ACKs with their piggybacked local status information, each instance in CoDeeN independently assesses the health of other nodes. This information is used by the redirector to determine which nodes are viable candidates for handling forwarded requests. Additionally, the UDP heartbeat facility has a mechanism by which a node can request a summary of the peer's health assessment. This mechanism is not used in normal operation, but is used for our central reporting system to observe overall trends. For example, by querying all CoDeeN nodes, we can determine which nodes are being avoided and which are viable.

2.2.2 HTTP/TCP Heartbeat

While the UDP-based heartbeat is useful for excluding some nodes, it cannot definitively determine node health, since it cannot test some of the paths that may lead to service failures. For example, we have experienced site administrators port filtering TCP connections, which can lead to UDP packets being exchanged without obstruction, but all TCP connections resulting in failure after failed retransmission attempts.

To augment our simple heartbeat, we also employ a tool to fetch pages over HTTP/TCP using a proxy. This tool, conceptually similar to the "wget" program [10], is instrumented to specify what fails when it cannot retrieve a page within the allotted time. Possible causes include socket allocation failure, slow/failed DNS lookup, incomplete connection setup, and failure to retrieve data from the remote system. The DNS resolver timing measurements from this tool are fed into the instance's local monitoring facilities. Since the fetch tool tests the proxying capabilities of the peers, we must also have "known good" web servers to use as origin servers. For this reason, each CoDeeN instance also includes a dummy web server that generates a noncacheable response page for incoming requests.

The local node picks one of its presumed live peers to act as the origin server, and iterates through all of the possible peers as proxies using the fetch tool. After one iteration, it determines which nodes were unable to serve the requested page. Those nodes are tested to see if they can serve a page from their own dummy server. These tests indicate whether a peer has global connectivity or any TCP-level connectivity at all.

Over time, all CoDeeN nodes will act as an origin server and a test proxy for this testing. We keep a history of the failed fetches for each peer, and combine this with the UDP-level heartbeats to determine if a node is viable for redirection. To allow for network delays and the possibility of the origin server becoming unavailable during one sweep, a node is considered bad if its failure count exceeds the other nodes by more than two. At current scale, the overhead for this iteration is tolerable. For much larger deployments, a hierarchical structure can limit the number of nodes actively communicating with each other.

2.3 Aggregate Information

Each CoDeeN proxy stores its local monitoring state as well as its peer summary to disk every 30 seconds, allowing offline behavior analysis as well as anomaly detection. The summary is also published and updated automatically on the CoDeeN central status page [16] every five minutes. These logs provide the raw data that we use in our analysis in Section 5. A sample log entry, truncated to fit in the column, is shown in Figure 2.

Most of the fields are the measurements that have been mentioned earlier, and the columns in the tabular output represent data about the other nodes in CoDeeN. Values in these lines are usually the counts in base-32 format, where 'w' represents 32. The exception is SysMxCPU, which is the percentage value divided by 10 and rounded up. Based on collected information through UDP heartbeat and HTTP tests, each redirector decides the "Liveness" for each CoDeeN node, indicating whether the local node considers that peer node to be viable.

In this particular example, this node is avoiding six of its peers, mostly because they have missed several UDP

```

FdTstHst: 0x0
ProxOptm: 36707
NodeOptm: 111788
LoadAvgs: 0.18 0.24 0.33
ReqsHrly: 5234 3950 0 788 1004 275 2616
DNSFails: 0.00
DNSTimes: 2.48
SysPtCPU: 2 2 1 3 2 4

Liveness: ..X.. ..X.. ..... .X.XX ..... ..X. ....
MissAcks: 10w00 00001 00000 0w066 00010 000v0 00020
LateAcks: 00000 00000 00000 00000 00000 00000 00000
NoFdAcks: 00000 00000 00000 00000 00000 00000 00000
VersProb: 00000 00000 00000 00000 00000 00000 00000
MaxLoads: 41022 11111 11141 20344 11514 14204 11111
SysMxCPU: 81011 11111 11151 10656 11615 15564 11111
WgetProx: 00w00 00100 00010 0w110 00000 000s0 00010
WgetTarg: 11w11 10301 01021 1w220 00111 101t0 11121

```

Figure 2: Sample monitoring log entry

ACKs. The eighth node, highlighted in boldface, is being avoided because it has a WgetTarg count of 3, indicating that it has failed the HTTP fetch test (with itself as the target) three times out of the past 32. More analysis on the statistics for node avoidance is presented in Section 5.

3 Security Problems

To make the system more useful and increase the amount of traffic we receive, we allow all CoDeeN nodes to act as “open” proxies, accepting requests from any client in the world. However, this choice also opens the doors to many security problems. In this section, we discuss some of the problems we encountered during the early development and testing of CoDeeN, and the measures we took to deal with these problems. For the purposes of discussion, we have broadly classified the problems into those dealing with spammers, bandwidth consumption, high request rates, content theft, and anonymity, though we realize that some problems can fall into multiple areas.

3.1 Spammers

The conceptually simplest category of CoDeeN abuser is the spammer, though the mechanisms for spamming using a proxy server are different from traditional spamming. We encountered three different approaches – SMTP tunnels, CGI/formmail POST requests, and IRC spamming. These mechanisms exist without the use of proxies, but gain a level of indirection via proxies, complicating investigation. When faced with complaints, the administrators of the affected system must cooperate with the proxy administrators to find the actual spammer’s IP address.

SMTP tunnels – Proxies support TCP-level tunneling via the CONNECT method, mostly to support end-to-end SSL behavior when used as firewalls. After the client specifies the remote machine and port number, the proxy creates a new TCP connection and forwards data in both directions. Our nodes disallow tunneling to port 25 (SMTP) to prevent facilitating open relay abuse, but continually receive such requests. The prevalence and magnitude of such attempts is shown in Figure 3. As a test, we directed these requests to local honey-pot SMTP servers.

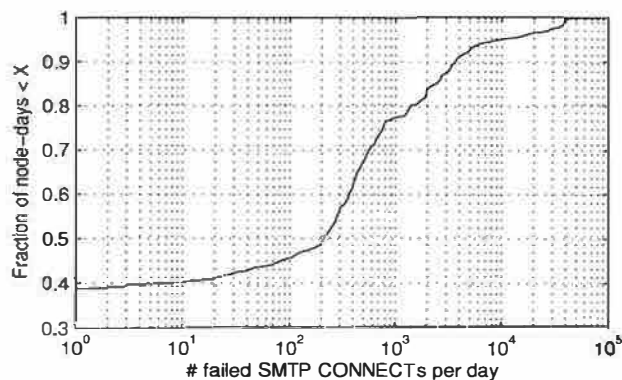


Figure 3: CONNECT activity for 38 nodes – Almost 40% of the samples show no activity, while 20% show over 1000 attempts/day. The maximum seen is over 90K attempts to one node in one day.

In one day, one of our nodes captured over 100K spam e-mails destined to 2,000,000 addresses. Another node saw traffic jump from 3,000 failed attempts per day to 30,000 flows in 5 minutes. This increase led to a self-inflicted denial-of-service when the local system administrator saw the activity spike and disconnected the PlanetLab node.

POST/formmail – Some web sites use a CGI program called *formmail* to allow users to mail web-based feedback forms to the site’s operators. Unfortunately, these programs often store the destination e-mail address in the form’s hidden input, relying on browsers to send along only the e-mail address specified in the form. Spammers abuse those scripts by generating requests with their victims’ e-mail addresses as the targets, causing the exploited site to send spam to the victim.

IRC – Spammers target IRC networks due to their weak authentication and their immediate, captive audience. Most proxies allow CONNECTs to ports above the protected port threshold of 1024, which affects IRC with its default port of 6667. IRC operators have developed their own open proxy blacklist [4], which checks IRC participant IP addresses for open proxies. We were alerted that CoDeeN was being used for IRC spamming, and found many of our nodes blacklisted. While the blacklists eliminate the problem for participating IRC networks, the collateral damage can be significant if other sites begin to refuse non-IRC traffic from blacklisted nodes.

3.2 Bandwidth Hogs

CoDeeN is hosted on PlanetLab nodes, with the hosts absorbing the bandwidth costs. Since most nodes are hosted at high-bandwidth universities, they attract people performing bulk data transfers. Due to lack of locality, such transfers provide no benefit to other CoDeeN users – they cause cache pollution and link congestion.

Webcam Trackers – Sites such as SpotLife.com provide a simple means to use digital cameras as auto-updating web cameras. This *subscription-based* service

allows the general public to broadcast their own “webcams”. We noticed heavy bandwidth usage of the SpotLife site, with individual IP addresses generating multiple image requests per second, far above the rate limits in the official SpotLife software. SpotLife claims to bundle their software with over 60% of digital cameras, and a community of high-rate downloaders has formed, to SpotLife’s consternation. These users clearly have enough bandwidth to access webcams directly, but use CoDeeN to mask their identity.

Cross-Pacific Downloads – CoDeeN nodes in Washington and California received very high bandwidth consumption with both source and destination located along the Eastern rim of Asia. The multi-megabyte downloads appeared to be for movies, though the reason that these clients chose a round-trip access across the Pacific Ocean is still not clear to us. A direct connection would presumably have much lower latency, but we suspect that these clients were banned from these sites, and required high-bandwidth proxies to access them effectively. Given the high international bandwidth costs in Asia, Western US proxies were probably easier to find.

Steganographers – While large cross-Pacific transfers were easy to detect in access logs, others were less obvious. This class had high aggregate traffic, spread across uniformly-sized, sub-megabyte files marked as GIFs and JPEGs. Large images sizes are not uncommon in broadband-rich countries such as South Korea, but some size variation is expected given the unpredictability of image compression. We downloaded a few of these large files and found that they generated only tiny images on-screen. From the URL names, we assume that these files contain parts of movies stuffed inside image files to hide their actual payload. Although there is existing research on steganography [18], we have not found the appropriate decryption tools to confirm our guess.

3.3 High Request Rates

TCP’s flow/congestion controls mitigate the damage that bulk transfers have on other CoDeeN users. In contrast, another class of users generated enough requests that we were concerned that CoDeeN might be implicated in a denial-of-service attack.

Password Crackers – We found an alarming number of clients using CoDeeN to launch dictionary attacks on Yahoo, often via multiple CoDeeN nodes. At one point, we were detecting roughly a dozen new clients per day. Since Yahoo can detect multiple failed attempts to a single account, these users try a single password across many accounts. The attacks appear to be for entertainment, since any victim will be random rather than someone known to the attacker. The problem, again, is that the requests appear to come from CoDeeN, and if Yahoo blocks the IP address, then other PlanetLab services are affected.

Google Crawlers – Like password crackers, we found a number of clients performing Google web/image searches on a series of sorted words. These were clearly mechanical processes working from a dictionary, and their requests were evenly spaced in time. We speculate that these clients are trying to populate their own search engines or perhaps build offline copies of Google.

Click-Counters – Ad servers count impressions for revenue purposes, and rarely do we see such accesses not tied to actual page views. The one exception we have seen is a game site called OutWar.com. Points are obtained when people click on a player’s “special link”, which delivers a Web page containing ad images. The system apparently counts hits of the player’s link instead of ad views, which seems to invite abuse. We have noticed a steady stream of small requests for these links, presumably from players inflating their scores.

3.4 Content Theft

The most worrisome abuse we witnessed on CoDeeN was what we considered the most sophisticated – unauthorized downloading of licensed content.

Licensed Content Theft – Universities purchase *address-authenticated* site licenses for electronic journals, limited to the IP ranges they own. PlanetLab’s acceptable use policies disallow accessing these sites, but CoDeeN unintentionally extended this access worldwide. We discovered this problem when a site contacted PlanetLab about suspicious activity. This site had previously experienced a coordinated attack that downloaded 50K articles. Unfortunately, such sites do not handle the *X-Forwarded-For* header that some proxies support to identify the original client IP address. Though this header can be forged, it can be trusted when *denying* access, assuming nobody would forge it to deny themselves access to a site.

Intra-domain Access – Many university Web pages are similarly restricted by IP address, but are scattered within the domain, making them hard to identify. For example, a department’s web site may intersperse department-only pages among publicly-accessible pages. Opportunities arise if a node receives a request for a local document, whether that request was received directly or was forwarded by another proxy.

3.5 Anonymity

While some people use proxies for anonymity, some anonymizers accessing CoDeeN caused us some concern. Most added one of more layers of indirection into their activities, complicating abuse tracking.

Request Spreaders – We found that CoDeeN nodes were being advertised on sites that listed open proxies and sold additional software to make testing and using proxies easier. Some sites openly state that open proxies can be used for bulk e-mailing, a euphemism for spam. Many of these sites sell software that spreads requests over a

collection of proxies. Our concern was that this approach could flood a single site from many proxies.

TCP over HTTP – Other request traffic suggested that some sites provided HTTP-to-TCP gateways, named *http2tcp*, presumably to bypass corporate firewalls. Other than a few archived Usenet messages on Google, we have not been able to find more information about this tool.

Non-HTTP Port 80 – While port 80 is normally reserved for HTTP, we also detected CONNECT tunnels via port 80, presumably to communicate between machines without triggering firewalls or intrusion detection systems. However, if someone were creating malformed HTTP requests to attack remote web sites, port 80 tunnels would complicate investigations.

Vulnerability Testing – We found bursts of odd-looking URLs passing through CoDeeN, often having the same URI portion of the URL and different host names. We found lists of such URLs on the Web, designed to remotely test known buffer overflow problems, URL parsing errors, and other security weaknesses. In one instance, these URLs triggered an intrusion detection system, which then identified CoDeeN as the culprit.

4 Protecting CoDeeN

Our guiding principle in developing solutions to address these security problems is to allow users at PlanetLab sites as much access to the Web as they would have without using a proxy, and to allow other users as much “safe” access as possible. To tailor access policies, we classify client IP addresses into three groups – those local to this CoDeeN node, those local to any site hosting a PlanetLab node, and those outside of PlanetLab. Note that our security concerns focus on how we handle possibly malicious client traffic, and not node compromise, which is outside the scope of this paper.

4.1 Rate Limiting

The “outside” clients face the most restrictions on using CoDeeN, limiting request types as well as resource consumption. Only their GET requests are honored, allowing them to download pages and perform simple searches. The POST method, used for forms, is disallowed. Since forms are often used for changing passwords, sending e-mail, and other types of interactions with side-effects, the restriction on POST has the effect of preventing CoDeeN from being implicated in many kinds of damaging Web interactions. For the allowed requests, both request rate and bandwidth are controlled, with measurement performed at multiple scales – the past minute, the past hour, and the past day. Such accounting allows short-term bursts of activity, while keeping the longer-term averages under control. Disallowing POST limits some activities, notably on e-commerce sites that do not use SSL/HTTPS. We are investigating mechanisms to determine which POST actions are reasonably safe, but as more transactions move

to secure sites, the motivation for this change diminishes.

To handle overly-aggressive users we needed some mechanism that could quickly be deployed as a stopgap. As a result, we added an explicit blacklist of client IP addresses, which is relatively crude, but effective in handling problematic users. This blacklist was not originally part of the security mechanism, but was developed when dictionary attacks became too frequent. We originally analyzed the access logs and blacklisted clients conducting dictionary attacks, but this approach quickly grew to consume too much administrative attention.

The problem with the dictionary attacks and even the vulnerability tests is that they elude our other tests and can cause problems despite our rate limits. However, both have fairly recognizable characteristics, so we used those properties to build a fairly simple signature detector. Requests with specific signatures are “charged” at a much higher rate than other rate-limited requests. We effectively limit Yahoo login attempts to about 30 per day, frustrating dictionary attacks. We charge vulnerability signatures with a day’s worth of traffic, preventing any attempts from being served and banning the user for a day.

Reducing the impact of traffic spreaders is more difficult, but can be handled in various ways. The most lenient approach, allowing any client to use multiple nodes such that the sum does not exceed the request rate, requires much extra communication. A stricter interpretation could specify that no client is allowed to use more than K proxies within a specified time period, and would be more tractable. We opt for a middle ground that provides some protection against abusing multiple proxies.

In CoDeeN, cache misses are handled by two proxies – one acting as the client’s forward proxy, and the other as the server’s reverse proxy. By recording usage information at both, heavy usage of a single proxy or heavy aggregate use can be detected. We forward client information to the reverse proxies, which can then detect clients using multiple forward proxies. While forwarding queries produces no caching benefit, forwarding them from outside users allows request rate accounting to include this case. So, users attempting to perform Yahoo dictionary attacks (which are query-based) from multiple CoDeeN nodes find that using more nodes does not increase the maximum number of requests allowed. With these changes, login attempts passed to Yahoo have dropped by a factor of 50 even as the number of attackers has tripled.

4.2 Privilege Separation

To address the issue of restricting access to content, we employ privilege separation, which works by observing that when a proxy forwards a request, the request assumes the privilege level of the proxy since it now has the proxy’s IP address. Therefore, by carefully controlling which proxies handle requests, appropriate access privileges can be maintained. The ideal solution for protect-

ing licensed content would be to insert an 'X-Forwarded-For' header, but it requires cooperation from the content site – checking whether both the proxy address and forwarded address are authorized. Although this is a simple change, there are some sites that do not handle the header. For such sites, content protection requires CoDeeN to identify what content is licensed and we take an approximate approach. Using Princeton's e-journal subscription list as a starting point, we extracted all host names and pruned them to coalesce similarly-named sites, merging `journal1.example.com` and `journal2.example.com` into just `example.com`. We do not precisely associate subscriptions with universities, since that determination would be constantly-changing and error-prone.

When accessing licensed content, we current only allow requests that preserve privilege. Clients must choose a CoDeeN forward proxy in their own local domain in order to access such content. These *local clients* are assumed to have the same privilege as the CoDeeN forward proxy, so this approach does not create additional exposure risks. These requests are sent directly to the content provider by the forward proxy, since using a reverse proxy would again affect the privilege level. All other client requests for licensed content currently receive error messages. Whether the local client can ultimately access the site is then a decision that the content provider makes using the CoDeeN node's IP address. Though we cannot guarantee the completeness of the subscription list, in practice this approach appears to work well. We have seen requests rejected by this filter, and we have not received any other complaints from content providers. In the future, when dealing with accesses to licensed sites, we may redirect clients from other CoDeeN sites to their local proxies, and direct all "outside" clients to CoDeeN proxies at sites without any subscriptions.

A trickier situation occurs when restricted content is hosted in the same domain as a CoDeeN node, such as when part of a university's Web site is restricted to only those within the university. Protecting these pages from outside exposure cannot use the coarse-grained blacklisting approach suitable for licensed content. Otherwise, entire university sites and departments would become inaccessible. To address this problem, we preserve the privilege of local clients, and de-escalate the privilege of remote clients. We determine if a request to `example.edu` originates locally at `example.edu`, and if so, the request is handled directly by the CoDeeN forward proxy. Otherwise, the request is forwarded to a CoDeeN node at another site, and thereby gets its privilege level dropped to that of the remote site through this "bouncing" process. To eliminate the exposure caused by forwarding a request to a site where it is local, we modify our forwarding logic – no request is forwarded to a CoDeeN proxy that has the same domain as the requested content.

Since our security mechanisms depend on comparing host names, we also disallow "outside" accesses to machines identified only by IP addresses. After implementing this approach, we found that some requests using numerical IP addresses were still being accepted. In the HTTP protocol, proxies receive requests that can contain a full URL, with host name, as the first request line. Additional header lines will also identify the host by name. We found some requests were arriving with differing information in the first line and in the Host header. We had not observed that behavior in any Web browser, so we assume such requests were custom-generated, and modified our redirector to reject such abnormal requests.

4.3 Effectiveness of the Solutions

We have received a handful of queries/complaints from system administrators at the local PlanetLab sites, and all but one have been false alarms. Most queries have been caused by system administrators or others using/testing the proxies, surfing through them, and then concluding that they are open proxies.

We have been using CoDeeN daily, and have found that the security restrictions have few effects for local users. Using non-Princeton nodes as our forward proxy, we have found that the restrictions on licensed sites can be overly strict at times. We expect that in the future, when we bounce such requests to completely unprivileged proxies, the special handling for those sites will not be noticeable. These bounced requests will obtain the privilege level of those proxies (i.e., no subscriptions), and will be able to access unrestricted portions of those sites. By changing the configuration information, we have also been able to use CoDeeN as an outside user would see it. Even on our high-speed links, the request rates limits have not impacted our daily browsing.

Restricting outside users from using POST does not appear to cause significant problems in daily use. Searches are commonly handled using the GET method instead of the POST method, and many logins are being handled via HTTPS/SSL, which bypasses the proxy. The most noticeable restrictions on outsiders using POST has been the search function on Amazon.com and some chat rooms. Over two months, local users have generated fewer than 300 POST requests, with the heaviest generator being software update checkers from Apple and Microsoft.

Our security measures have caused some confusion amongst malicious users, and they could not figure out whether or not CoDeeN is a network of real "open" proxies. We routinely observe clients testing proxies and then generating requests at very high rates, sometimes exceeding 50K reqs/hour. However, rarely do CoDeeN nodes see more than 20K valid reqs/hour. Some clients have generated over a million unsuccessful requests in stretches lasting longer than a day.

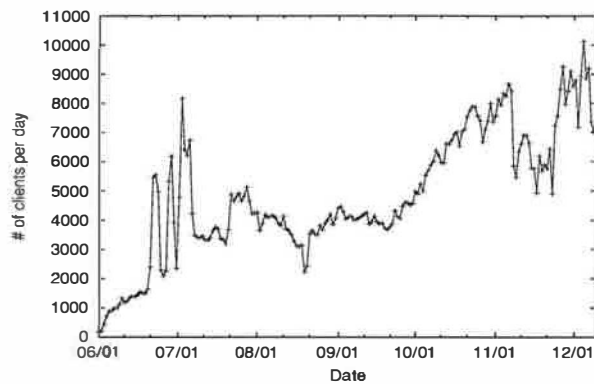


Figure 4: Daily Client Population (Unique IP) on CoDeeN

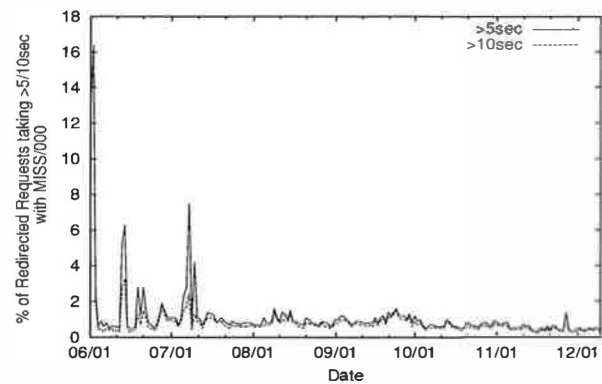


Figure 6: Percentage of Non-serviced Redirected Requests

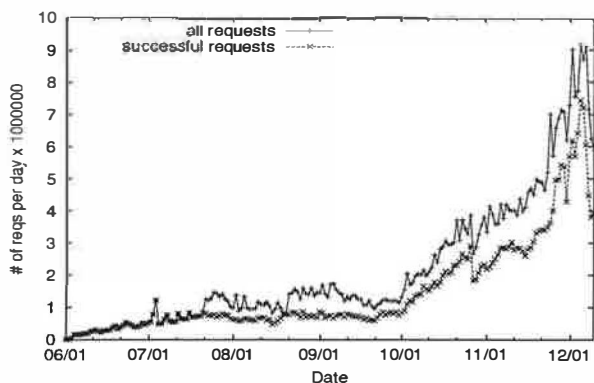


Figure 5: Daily Requests Received on CoDeeN

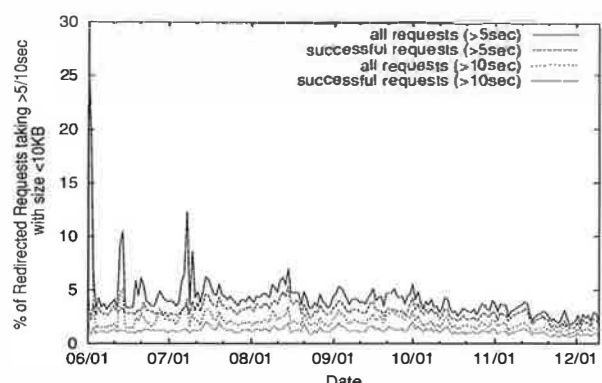


Figure 7: Percentage of Redirected Requests (< 10KB)

5 Results

In this section, we analyze the data we collected during six months of CoDeeN's operation. These results not only show the status of CoDeeN over time, but also provide insights into the monitoring and security measures.

5.1 Traffic

Since starting our public beta test at the end of May, the number of unique IP addresses used to access CoDeeN has passed 500,000, with the daily values shown in Figure 4. Some of these clients appear to be human, while others are programs that interact with proxies. Now, our daily traffic regularly exceeds 7,000 unique IPs.

The daily traffic served by CoDeeN now hovers above more than 4 million requests, and peaks at 7 million, as seen in Figure 5. The total count of daily requests, including those that are rejected, is approaching 7 million per day and peaks at 9 million. We began logging rejected requests in late July, so earlier figures are not available.

5.2 Response Performance

Since reliability has been one of the main thrusts of our current work, the response time behavior of CoDeeN is largely a function of how well the system performs in

avoiding bad nodes. In the future, we may work towards optimizing response time by improving the redirector logic, but that has not been our focus to date.

The results of our efforts to detect/avoid bad nodes can be seen in Figure 6, which shows requests that did not receive any service within specific time intervals. When this occurs, the client is likely to stop the connection or visit another page, yielding an easily-identifiable access log entry (MISS/000). These failures can be the result of the origin server being slow or a failure within CoDeeN. The trend shows that both the magnitude and frequency of the failure spikes are decreasing over time. Our most recent change, DNS failure detection, was added in late August, and appears to have yielded positive results.

Since we cannot "normalize" the traffic over CoDeeN, other measurements are noisier, but also instructive. Figure 7 shows the fraction of small/failed responses that take more than a specific amount of time. Here, we only show redirected requests, which means they are not serviced from the forward proxy cache. By focusing on small responses, we can remove the effects of slow clients downloading large files. We see a similar trend where the failure rate decreases over time. The actual overall response

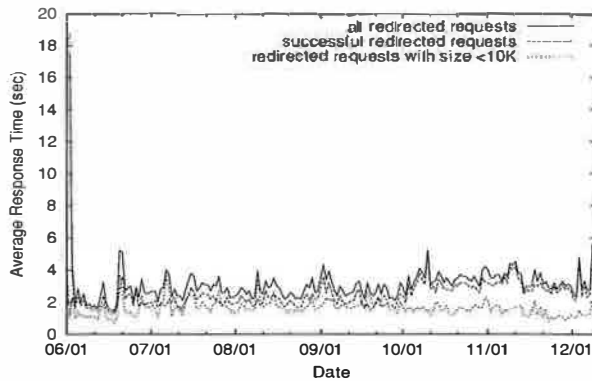


Figure 8: Average Response Time of Redirected Requests

times for successful requests, shown in Figure 8, has a less interesting profile. After a problematic beginning, responses have been relatively smooth. As seen from Figure 5, since the beginning of October, we have received a rapidly increasing number of requests on CoDeeN, and consequently, the average response time for all requests slightly increases over time. However, the average response time for small files is steady and keeps decreasing. This result is not surprising, since we have focused on reducing failures rather than reducing success latency.

5.3 Node Stability

The distributed node health monitoring system employed by CoDeeN, described in Section 2.2, provides data about the dynamics of the system and insight into the suitability of our choices regarding monitoring. One would expect that if the system is extremely stable and has few status changes, an active monitoring facility may not be very critical and probably just increases overhead. Conversely, if most failures are short, then avoidance is pointless since the health data is too stale to be useful. Also, the rate of status changes can guide the decisions regarding peer group size upper bounds, since larger groups will require more frequent monitoring to maintain tolerable staleness.

Our measurements confirm our earlier hypothesis about the importance of taking a monitoring and avoidance approach. They show that our system exhibits fairly dynamic liveness behavior. Avoiding bad peers is essential and most failure time is in long failures so avoidance is an effective strategy. Figure 9 depicts the stability of the CoDeeN system with 40 proxies from four of our CoDeeN redirectors' local views. We consider the system to be stable if the status of all 40 nodes is unchanged between two monitoring intervals. We exclude the cases where the observer is partitioned and sees no other proxies alive. The x -axis is the stable period length in seconds, and the y -axis is the cumulative percentage of total time. As we can see, these 4 proxies have very similar views. For about 8% of the time, the liveness status of all proxies changes

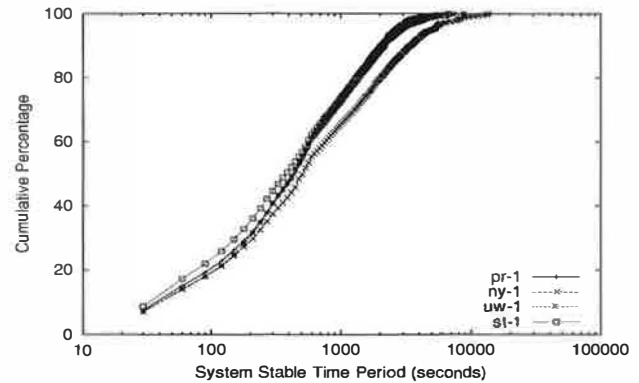
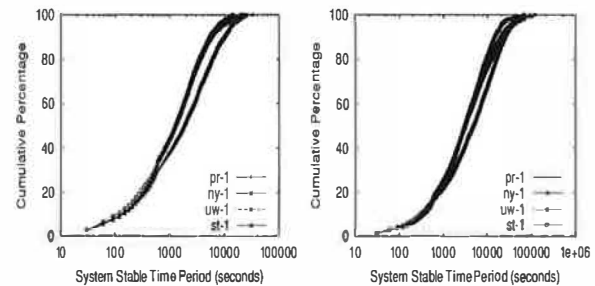


Figure 9: System Stability View from Individual Proxies



(a) Divided into 2 Groups

(b) Divided into 4 Groups

Figure 10: System stability for smaller groups

every 30 seconds (our measurement interval). In Table 1, we show the 50th and the 90th percentiles of the stable periods. For 50% of time, the liveness status of the system changes at least once every 6-7 minutes. For 90% of time, the longest stable period is about 20-30 minutes. It shows that in general, the system is quite dynamic – more than what one would expect from few joins/exits.

The tradeoff between peer group size and stability is an open area for research, and our data suggests, quite naturally, that stability increases as group size shrinks. The converse, that large groups become less stable, implies that large-scale peer-to-peer systems will need to sacrifice latency (via multiple hops) for stability. To measure the stability of smaller groups, we divide the 40 proxies into 2 groups of 20 and then 4 groups of 10 and measure group-wide stability. The results are shown in Figure 10 and also in Table 1. As we can see, with smaller groups, the stability improves with longer stable periods for both the 50th and 90th percentiles.

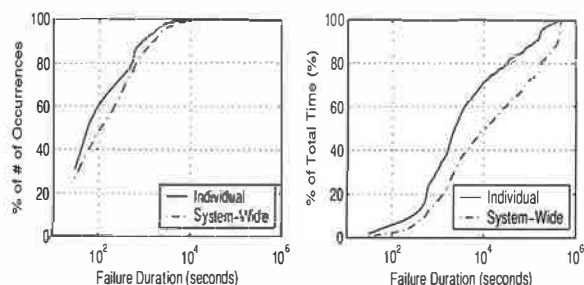
The effectiveness of monitoring-based avoidance depends on the node failure duration. To investigate this issue, we calculate node avoidance duration as seen by each node and as seen by the sum of all nodes. The distribution of these values is shown in Figure 11, where “Individ-

	40-node		2 × 20-node		4 × 10-node	
	50%	90%	50%	90%	50%	90%
pr-1	445	2224	1345	6069	3267	22752
ny-1	512	3451	1837	10020	4804	25099
uw-1	431	2085	1279	5324	3071	19579
st-1	381	2052	1256	5436	3008	14334

Table 1: System Stable Time Period (Seconds)

Site	Fetch	Miss ACKs	Node Down	Late ACKs	DNS
pr-1	6.2	18.3	29.6	13.6	32.1
ny-1	4.7	16.1	31.7	14.0	33.9
uw-1	10.4	16.8	30.0	12.8	29.7
st-1	5.0	14.7	27.2	15.4	34.3

Table 2: Average Percentage of Reasons to Avoid A Node



(a) CDF by # of Occurrences

(b) CDF by Total Time

Figure 11: Node Failure Duration Distribution. Failures spanning across a system-wide downtime are excluded from this measurement, so that it only includes *individual* node failures. Also, due to the interval of node monitoring, it may take up to 40 seconds for a node to be probed by another nodes, thus failures that last a shorter time might be neglected.

ual” represents the distribution as seen by each node, and “System-Wide” counts a node as failed if all nodes see it as failed. By examining the durations of individual failure intervals, shown in Figure 11a, we see that most failures are short, and last less than 100 seconds. Only about 10% of all failures last for 1000 seconds or more. Figure 11b shows the failures in terms of their contribution to the total amount of time spent in failures. Here, we see that these small failures are relatively insignificant – failures less than 100 seconds represent 2% of the total time, and even those less than 1000 seconds are only 30% of the total. These measurements suggest that node monitoring can successfully avoid the most problematic nodes.

5.4 Reasons to Avoid a Node

Similar to other research on peer-to-peer systems, we initially assumed that churn, the act of nodes joining and leaving the system, would be the underlying cause of staleness-related failures. However, as can be seen from the stability results, failure occurs at a much greater rate than churn. To investigate the root causes, we gather the logs from 4 of redirectors and investigate what causes nodes to switch from viable to avoided. Therefore, our counts also take time into account, and a long node failure receives more weight. We present each reason category with a non-negligible percentage in Table 2. We find that the underlying cause is roughly common across nodes – mainly dominated by DNS-related avoidance and many nodes down for long periods, followed by missed

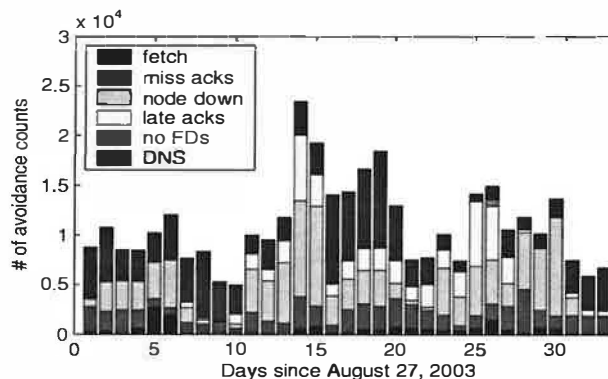


Figure 12: Daily counts of avoidance on ny-1 proxy

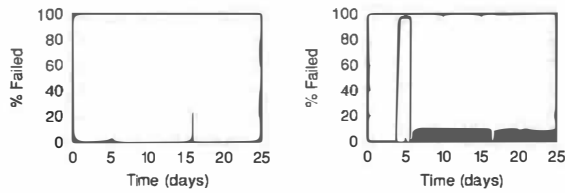
ACKs. Even simple overload, in the form of late ACKs, is a significant driver of avoidance. Finally, the HTTP fetch helper process can detect TCP-level or application-level connectivity problems.

In terms of design, these measurements show that a UDP-only heartbeat mechanism will significantly underperform our more sophisticated detection. Not only are the multiple schemes useful, but they are complementary. Variation occurs not only across nodes, but also within a node over a span of multiple days. The data for the ny-1 node, calculated on a daily basis, is shown in Figure 12.

5.5 DNS behaviors

As described earlier, during our HTTP fetch tests, we measure the time of local DNS lookups. When local name servers are having problems, DNS lookups can take many seconds to finish, despite usually taking only a few milliseconds. We further investigate how DNS lookups behave on each proxy by looking at DNS failure rates and average response time for successful queries. If a DNS lookup takes longer than 5 seconds, we regard it as a DNS failure, since this value is the resolver’s default timeout.

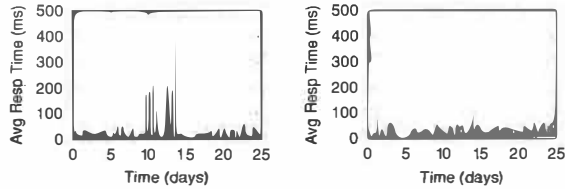
Figures 13 and 14 show the DNS failure rates and DNS average lookup time for successful queries on 2 of our sampling proxies, ny-1 on east coast and st-1 on west coast. DNS lookup time is usually short (generally well below 50ms), but there are spikes of 50-100ms. Recall that these lookups are only for the controlled set of the intra-CoDeeN “fetch” lookups. Since these mappings are stable, well-advertised, and cacheable, responses should be fast for well-behaved name servers. Anything more than tens of milliseconds implies the local nameservers



(a) ny-1 proxy

(b) st-1 proxy

Figure 13: DNS lookup failure at different proxies



(a) ny-1 proxy

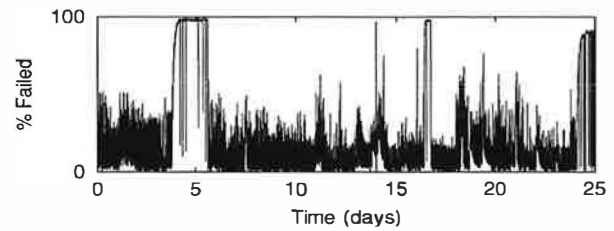
(b) st-1 proxy

Figure 14: DNS lookup time at different proxies

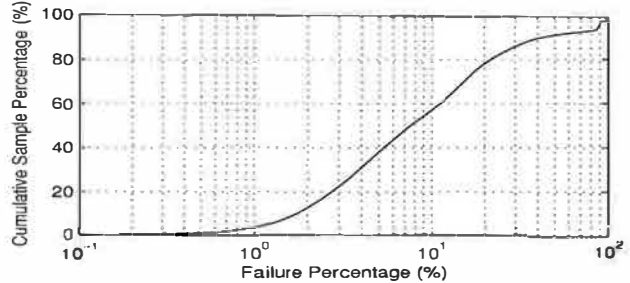
are having problems. These statistics also help to reveal some major problems. For example, the st-1 proxy has a period of 100% DNS failure rate, which is due to the name server disappearing. The problem was resolved when the node's resolv.conf file was manually modified to point to working name servers.

Though DNS failure rates on individual proxies are relatively low, the combined impact of DNS failures on web content retrieval is alarming. Downloading a common web page often involves fetching the attached objects such as images, and the corresponding requests can be forwarded to different proxies. Supposing an HTTP session involves 20 proxies, Figure 15 shows the probability of incurring at least one DNS failure in the session. From the cumulative distribution we can see that for more than 40% of time we have DNS failure probability of at least 10%, which would lead to a pretty unpleasant surfing experience if we did not avoid nodes with DNS problems. Note that these problems often appear to stem from factors beyond our control – Figure 16 shows a DNS nameserver exhibiting periodic failure spikes. Such spikes are common across many nameservers, and we believe that they reflect a cron-initiated process running on the nameserver.

To avoid such problems, we have taken two approaches to reduce the impact of DNS lookups in CoDeeN. The first is a change in redirector policy that is intended to send all requests from a single page to the same reverse proxy node. If a request contains a valid "referer" header, it is used in the hashing process instead of the URL. If no such header exists, the last component of the URL is omitted when hashing. Both of these techniques will tend to send all requests from a single page to the same reverse proxy.



(a) Aggregate DNS failure rate over 25 days



(b) Cumulative distribution DNS failure rate

Figure 15: DNS failure rate of 20 nodes, i.e. the probability of at least one node having DNS difficulty. The abnormal peak around day 5 in (a) is caused by the same peak in Figure 13(b). Thus when computing the cumulative distribution in (b) we only considered the last 15 days.

Not only will this result in fewer DNS lookups, but it will also exploit persistent connections in HTTP. The second modification to reduce the problems stemming from DNS is a middleware DNS brokering service we have developed, called CoDNS. This layer can mask local DNS resolver failure by using remote resolvers, and is described elsewhere [17].

5.6 Requests Rejected for Security Reasons

In Section 3, we explored the idea of rejecting requests that could cause security problems or abuse system resources. Figure 17 shows a snapshot of the statistics about various reasons for rejecting requests. Three major reasons include clients exceeding the maximum rate, requests using methods other than GET and requests with no host field, indicating non-standard browsers. Most of the time, these three comprise more than 80% of the rejected traffic. The query count represents the number of bandwidth capped CGI queries which include all sorts of malicious behaviors previously mentioned. Disallowed CONNECTs and POSTs indicate attempts to send spam through our system. CONNECTs alone constitute, on the average, over 5% and sometimes 30% of all rejected requests. From this graph, we can get an idea of how many scavenging attempts are being made through the open proxies like CoDeeN.

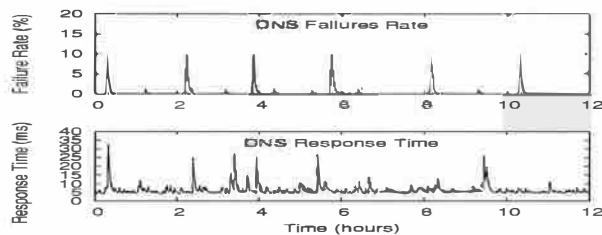


Figure 16: DNS failures, response times for the Stanford proxy

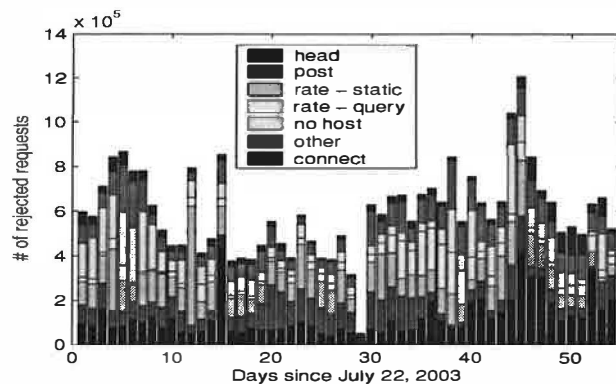


Figure 17: Daily counts of rejected requests on CoDeeN

We assume most of this traffic is being generated automatically by running some custom programs. We are now studying how to identify these malicious programs versus normal human users and innocuous programs like web crawlers, in order to provide an application-level QoS depending on client classification.

6 Related Work

Similar to CoDeeN, peer-to-peer systems [20, 22, 24] also run in a distributed, unreliable environment. Nodes join or depart the system from time to time, and node failures can also happen often. Besides maintaining a membership directory, these systems typically follow a retry and failover scheme to deal with failing nodes while routing to the destinations. Although practically, these trials can be expected by peer-to-peer system users, the extra delays in retrying different next hops can cause latency problems. For latency-sensitive applications implemented on a peer-to-peer substrate, multiple hops or trials in each operation become even more problematic [7]. The Globe distribution network also leverages hierarchy and location caching to manage mobile objects [3]. To address multiple-hop latency, recent research has started pushing more membership information into each node in a peer-to-peer system to achieve one-hop lookups [12, 21]. In this regard, similar arguments can be made that each node could monitor the status of other nodes.

Some researchers have used Byzantine fault tolerant

approaches to provide higher reliability and robustness than fail-stop assumptions provide [1, 5]. While such schemes, including state machine replication in general, may seem appealing for handling failing nodes in CoDeeN, the fact that origin servers are not under our control limits their utility. Since we cannot tell that an access to an origin server is idempotent, we cannot issue multiple simultaneous requests for one object due to the possibility of side-effects. Such an approach could be used among CoDeeN's reverse proxies if the object is known to be cached.

In the cluster environment, systems with a front end [11] can deploy service-specific load monitoring routines in the front end to monitor the status of server farms and decide to avoid failing nodes. These generally operate in a tightly-coupled environment with centralized control. There are also general cluster monitoring facilities that can watch the status of different nodes, such as the Ganglia tools [9], which have already been used on PlanetLab. We can potentially take advantage of Ganglia to collect system level information. However, we are also interested in application-level metrics such as HTTP/TCP connectivity, and some of resources such as DNS behaviors that are not monitored by Ganglia.

Cooperative proxy cache schemes have been previously studied in the literature [6, 19, 25, 27], and CoDeeN shares many similar goals. However, to the best of our knowledge, the only two deployed systems have used the Harvest-like approach with proxy cache hierarchies. The main differences between CoDeeN and these systems are in the scale, the nature of who can access, and the type of service provided. Neither system uses open proxies. The NLNR Global Caching Hierarchy [15] operates ten proxy caches that only accept requests from other proxies and one end-user proxy cache that allows password-based access after registration. The JANET Web Cache Service [14] consists of 17 proxies in England, all of which are accessible only to other proxies. Joining the system requires providing your own proxy, registering, and using an access control list to specify which sites should not be forwarded to other caches. Entries on this list include electronic journals.

A new Akamai-like system, CoralCDN [8], is in the process of being deployed. Bad nodes are avoided by DNS-based redirection, sometimes using an explicit UDP RPC for status checking.

7 Conclusion

In this paper, we present our experience with a continuously running prototype CDN on PlanetLab. We describe our reliability mechanisms that assess node health and prevent failing nodes from disrupting the operation of the overall system. We also discuss our security mechanisms that protect nodes from being exploited and from

being implicated in malicious activities. The intentional dual use of CoDeeN both as a CDN and as an open proxy network and the resource competition on Planet-Lab nodes make it a very valuable testbed. We believe that future services, especially peer-to-peer systems, will require similar mechanisms as more services are deployed on non-dedicated distributed systems, and as their interaction with existing protocols and systems increases.

Our distributed monitoring facilities prove to be effective at detecting and thus avoiding failing or problematic nodes. The net benefit is robustness against component disruptions and improved response latency. Although some of the aspects of these facilities seem application-specific, they are not confined to CDN services. Other latency-sensitive services running in a non-dedicated distributed environment can potentially benefit from them, since they also need to do extra reliability checks. Our experiences also reveal that reliability-induced problems occur almost two orders of magnitude more frequently than node joins/leaves, which makes active monitoring necessary and important for other systems such as peer-to-peer.

Our security measures consist of classification, rate limiting, and privilege separation. They provide a model for other Web-accessible services. For example, some of the security mechanisms we are developing are suitable for ISPs to deploy on their own networks to detect misbehaving customers before problems arise. Other systems that allow open access to Web resources may face similar situations, and may be able to adopt similar mechanisms.

Our experiences with CoDeeN and the data we have obtained on availability can serve as a starting point for designers of future systems. We demonstrate that effective monitoring is critical for system proper operation, and security measures are important for preventing the system from being abused.

Acknowledgments

This research is supported in part by DARPA contract F30602-00-2-0561. We thank our shepherd, Atul Adya, for his guidance and helpful input. We also thank our anonymous reviewers for their valuable comments on improving this paper.

References

- [1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Dec. 2002.
- [2] Akamai. Content Delivery Network. <http://www.akamai.com>.
- [3] A. Baggio, G. Ballintijn, M. van Steen, and A. S. Tanenbaum. Efficient tracking of mobile objects in Globe. *The Computer Journal*, 44(5):340–353, 2001.
- [4] BOPM. Blitzed Open Proxy Monitor. <http://www.blitzed.org/bopm/>.
- [5] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.
- [6] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.
- [7] R. Cox, A. Muthitacharoen, and R. Morris. Serving dns using chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, 2002.
- [8] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. In *Proceedings of the 1st Symposium on Networked System Design and Implementation (NSDI '04)*, 2004.
- [9] Ganglia. <http://ganglia.sourceforge.net>.
- [10] GNU wget. <http://www.gnu.org/software/wget/wget.html>.
- [11] G. Goldszmidt and G. Hunt. Netdispatcher: A tcp connection router. IBM Research White Paper.
- [12] A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, 2003.
- [13] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the 21st Symposium on Principles of Distributed Computing (PODC)*, 2002.
- [14] JANET Web Cache Service. <http://www.wcache.ja.net>.
- [15] National Laboratory for Applied Network Research (NLNR). Ircache project. <http://www.ircache.net/>.
- [16] Network Systems Group, Princeton University. CoDeeN—A CDN on PlanetLab. <http://codeen.cs.princeton.edu>.
- [17] K. Park, Z. Wang, V. Pai, and L. Peterson. CoDNS: Masking DNS delays via cooperative lookups. Technical Report TR-690-04, Princeton University Computer Science Department, Feb. 2004.
- [18] N. Provos and P. Honeyman. Detecting steganographic content on the internet. In *ISOC NDSS'02*, Feb. 2002.
- [19] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: cooperative proxy caching over a wide-area network. *Computer Networks and ISDN Systems*, 30(22–23):2253–2259, 1998.
- [20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01*, Aug. 2001.
- [21] R. Rodrigues, B. Liskov, and L. Shira. The design of a robust peer-to-peer system. In *Tenth ACM SIGOPS European Workshop*, 2002.
- [22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.
- [23] Speedera. <http://www.speedera.com>.
- [24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM 2001*, San Diego, California, Aug. 2001.
- [25] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the internet. In *International Conference on Distributed Computing Systems*, pages 273–284, 1999.
- [26] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.
- [27] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, 1999.

Building Secure High-Performance Web Services with OKWS

Maxwell Krohn, *MIT Computer Science and AI Laboratory*

krohn@csail.mit.edu

Abstract

OKWS is a toolkit for building fast and secure Web services. It provides Web developers with a small set of tools that has proved powerful enough to build complex systems with limited effort. Despite its emphasis on security, OKWS shows performance improvements compared to popular systems: when servicing fully dynamic, non-disk-bound database workloads, OKWS's throughput and responsiveness exceed that of Apache 2 [3], Flash [23] and Haboob [44]. Experience with OKWS in a commercial deployment suggests it can reduce hardware and system management costs, while providing security guarantees absent in current systems.

1 Introduction

Most dynamic Web sites today maintain large server-side databases, to which their users have limited access via HTTP interfaces. Keeping this data hidden and correct is critical yet difficult. Indeed, headlines are replete with stories of the damage and embarrassment remote attackers can visit on large Web sites.

Most attacks against Web sites exploit weaknesses in popular Web servers or bugs in custom application-level logic. In practice, emphasis on rapid deployment and performance often comes at the expense of security.

Consider the following example: Web servers typically provide Web programmers with powerful and generic interfaces to underlying databases and rely on coarse-grained database-level permission systems for access control. Web servers also tend to package logically separate programs into one address space. If a particular Web site serves its *search* and *newsletter-subscribe* features from the same machine, a bug in the former might allow a malicious remote client to select all rows from a table of subscribers' email addresses. In general, anything from a buffer overrun to an unexpected escape sequence can expose private data to an attacker. Moreover, few practical isolation schemes exist aside from running different services on different machines. As a result, a flaw in one service can ripple through an entire system.

To plug the many security holes that plague existing

Web servers, and to limit the severity of unforeseen problems, we introduce OKWS, the OK Web Server. Unlike typical Web servers, OKWS is specialized for dynamic content and is not well-suited to serving files from disk. It relies on existing Web servers, such as Flash [23] or Apache [3], to serve images and other static content. We argue (in Section 5.4) that this separation of static and dynamic content is natural and, moreover, contributes to security.

What OKWS does provide is a simple, powerful, and secure toolkit for building dynamic content pages (also known as *Web services*). OKWS enforces the natural principle of *least privilege* [27] so that those aspects of the system most vulnerable to attack are the least useful to attackers. Further, OKWS separates privileges so that the different components of the system distrust each other. Finally, the system distrusts the Web service developer, presuming him a sloppy programmer whose errors can cause significant damage. Though these principles are not novel, Web servers have not generally incorporated them.

Using OKWS to build Web services, we show that compromises among basic security principles, performance, and usability are unnecessary. To this effect, the next section surveys and categorizes attacks on Web servers, and Section 3 presents simple design principles that thwart them. Section 4 discusses OKWS's implementation of these principles, and Section 5 argues that the resulting system is practical for building large systems. Section 6 discusses the security achieved by the implementation, and Section 7 analyzes its performance, showing that OKWS's specialization for dynamic content helps it achieve better performance in simulated dynamic workloads than general purpose servers.

2 Brief Survey of Web Server Bugs

To justify our approach to dynamic Web server design, we briefly analyze the weaknesses of popular software packages. Our goal is to represent the range of bugs that have arisen in practice. Historically, attackers have exploited almost all aspects of conventional Web servers, from core components and scripting language exten-

sions to the scripts themselves. The conclusion we draw is that a better design—as opposed to a more correct implementation—is required to get better security properties.

In our survey, we focus on the Apache [3] server due to its popularity, but the types of problems discussed are common to all similar Web servers, including IBM WebSphere [14], Microsoft IIS [19] and Zeus [47].

2.1 Apache Core and Standard Modules

There have been hundreds of major bugs in Apache's core and in its standard modules. They fit into the following categories:

Unintended Data Disclosure. A class of bugs results from Apache delivering files over HTTP that are supposed to be private. For instance, a 2002 bug in Apache's `mod_dav` reveals source code of user-written scripts [42]. A recent discovery of leaked file descriptors allows remote users to access sensitive log information [7]. On Mac OS X operating systems, a local find-by-content indexing scheme creates a hidden yet world-readable file called `.FBCIndex` in each directory indexed. Versions of Apache released in 2002 expose this file to remote clients [41]. In all cases, attackers can use knowledge about local configuration and custom-written application code to mount more damaging attacks.

Buffer Overflows and Remote Code Execution. Buffer overflows in Apache and its many modules are common. Unchecked boundary conditions found recently in `mod_alias` and `mod_rewrite` regular expression code allow local attack [39]. In 2002, a common Apache deployment with OpenSSL had a critical bug in client key negotiation, allowing remote attackers to execute arbitrary code with the permissions of the Web server. The attacking code downloads, compiles and executes a program that seeks to infect other machines [36].

There have been less-sophisticated attacks that resulted in arbitrary remote code execution. Some Windows versions of Apache execute commands in URLs that follow pipe characters (`'|'`). A remote attacker can therefore issue the command of his choosing from an unmodified Web browser [40]. On MS-DOS-based systems, Apache failed to filter out special device names, allowing carefully-crafted HTTP POST requests to execute arbitrary code [43]. Other problems have occurred when site developers call Apache's `htdigest` utility from within CGI scripts to manage HTTP user authentication [6].

Denial of Service Attacks. Aside from TCP/IP-based DoS attacks, Apache has been vulnerable to a number of application-specific attacks. Apache versions released in 2003 failed to handle error conditions on certain “rarely used ports,” and would stop servicing incoming connections as a result [38]. Another 2003 release allowed local configuration errors to result in infinite redirection loops [8]. In some versions of Apache, attackers could exhaust Apache's heap simply by sending a large sequence of linefeed characters [37].

2.2 Scripting Extensions to Apache

Apache's security worsens considerably when compiled with popular modules that enable dynamically-generated content such as PHP [25]. In the past two years alone, at least 13 critical buffer overruns have been found in the PHP core, some of which allowed attackers to remotely execute arbitrary code [9, 28]. In six other cases, faults in PHP allowed attackers to circumvent its application level *chroot*-like environment, called “Safe Mode.” One vulnerability exposed `/etc/passwd` via `posix_getpwnam` [5]. Another allowed attackers to write PHP scripts to the server and then remotely execute them; this bug persisted across multiple releases of PHP intended as fixes [35].

Even if a correct implementation of PHP were possible, it would still provide Web programmers with ample opportunity to introduce their own vulnerabilities. A canonical example is that beginning PHP programmers fail to check for sequences such as “`..`” in user input and therefore inadvertently allow remote access to sensitive files higher up in the file system hierarchy (e.g., `../../../../etc/passwd`). Similarly, PHP scripts that embed unescaped user input inside SQL queries present openings for “SQL Injection.” If a PHP programmer neglects to escape user input properly, a malicious user can turn a benign `SELECT` into a catastrophic `DELETE`.

The PHP manual does state that PHP scripts might be separated and run as different users to allow for privilege separation. In this case, however, PHP could not run as an Apache module, and the system would require a new PHP process forked for every incoming connection. This isolation strategy is at odds with performance.

3 Design

If we assume that bugs like the ones discussed above are inevitable when building a large system, the best remedy is to limit the effectiveness of attacks when they occur. This section presents four simple guidelines for protecting sensitive site data in the worst-case scenario, in which

an adversary remotely gains control of a Web server and can execute arbitrary commands with the Web server's privileges. We also present OKWS's design, which follows the four security guidelines without sacrificing performance.

Throughout, we assume a cluster of Web servers and database machines connected by a fast, firewalled LAN. Site data is cached at the Web servers and persistently stored on the database machines. The primary security goals are to prevent intrusion and to prevent unauthorized access to site data.

3.1 Practical Security Guidelines

(1) *Server processes should be chrooted.* After compromising a server process, most attackers will try to gain control over the entire server machine, possibly by installing "back doors," learning local passwords or private keys, or probing local configuration files for errors. At the very least, a compromised Web server should have no access to sensitive files or directories. Moreover, an OS-level jail ought to hide all `setuid` executables from the Web server, since many privilege escalation attacks require such files (examples include the *ptrace* and *bind* attacks mentioned in [17]). Privilege escalation is possible without `setuid` executables but requires OS-level bugs or race conditions that are typically rarer.

An adversary can still do damage without control of the Web server machine. The configuration files, source files, and binaries that correspond to the currently running Web server contain valuable hints about how to access important data. For instance, PHP scripts often include the username and plaintext password used to gain access to a MySQL database. OS-enforced policy ought to hide these files from running Web servers.

(2) *Server processes should run as unprivileged users.* A compromised process running as a privileged user can do significant damage even from within a *chrooted* environment. It might bind to a well-known network port. It might also interfere with other system processes, especially those associated with the Web server: it can trace their system calls or send them signals.

(3) *Server processes should have the minimal set of database access privileges necessary to perform their task.* Separate processes should not have access to each other's databases. Moreover, if a Web server process requires only row-wise access to a table, an adversary who compromises it should not have the authority to perform operations over the entire table.

(4) *A server architecture should separate indepen-*

dent functionality into independent processes. An adversary who compromises a Web server can examine its in-memory data structures, which might contain soft state used for user session management, or possibly secret tokens that the Web server uses to authenticate itself to its database. With control of a Web server process, an adversary might hijack an existing database connection or establish a new one with the authentication tokens it acquired. Though more unlikely, an attacker might also monitor and alter network traffic entering and exiting a compromised server.

The important security principle here is to limit the types of data that a single process can access. Site designers should partition their global set of site data into small, self-contained subsets, and their Web server ought to align its process boundaries with this partition.

If a Web server implements principles (1) through (4), and if there are no critical kernel bugs, an attacker cannot move from vulnerable to secure parts of the system. By incorporating these principles, a Web server design assumes that processes will be compromised and therefore prevents uncompromised processes from performing unsafe operations, even when extended by careless Web developers. For example, if a server architecture denies a successful attacker access to `/etc/passwd`, then a programmer cannot inadvertently expose this file to remote clients. Similarly, if a successful attacker cannot arbitrarily access underlying databases, then even a broken Web script cannot enable SQL injection attacks.

3.2 OKWS Design

We designed OKWS with these four principles in mind. OKWS provides Web developers with a set of libraries and helper processes so they can build Web services as independent, stand-alone processes, isolated almost entirely from the file system. The core libraries provide basic functionality for receiving HTTP requests, accessing data sources, composing an HTML-formatted response, responding to HTTP requests, and logging the results to disk. A process called OK launcher daemon, or *okld*, launches custom-built services and relaunches them should they crash. A process called OK dispatcher, or *okd*, routes incoming requests to appropriate Web services. A helper process called *pubd* provides Web services with limited read access to configuration files and HTML template files stored on the local disk. Finally, a dedicated logger daemon called *oklogd* writes log entries to disk. Figure 1 summarizes these relationships.

This architecture allows custom-built Web services to meet our stated design goals:

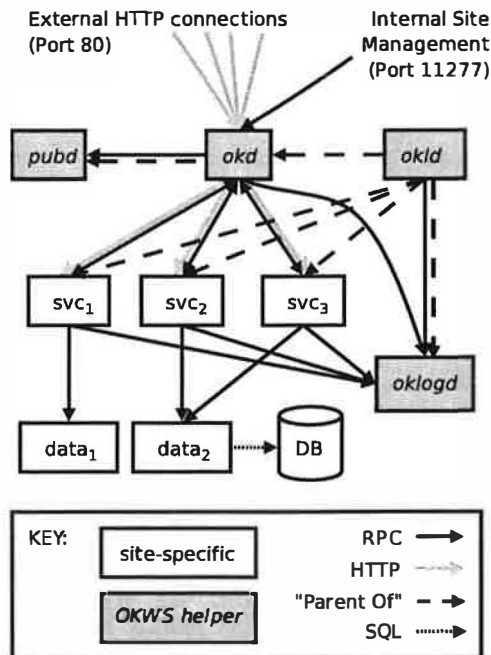


Figure 1: Block diagram of an OKWS site setup with three Web services (SVC_1, SVC_2, SVC_3) and two data sources ($data_1, data_2$), one of which ($data_2$) is an OKWS database proxy.

- (1) OKWS *chroots* all services to a remote jail directory. Within the jail, each process has just enough access privileges to read shared libraries upon startup and to dump core upon abnormal termination. The services otherwise never access the file system and lack the privileges to do so.
- (2) Each service runs as a unique non-privileged user.
- (3) OKWS interposes a structured RPC interface between the Web service and the database and uses a simple authentication mechanism to align the partition among database access methods with the partition among processes.
- (4) Each Web service runs as a separate process. The next section justifies this choice.

3.3 Process Isolation

Unlike the other three principles, the fourth, of process isolation, implies a security and performance tradeoff since the most secure option—one Unix process per *external user*—would be problematic for performance. OKWS's approach to this tradeoff is to assign one Unix process per *service*; we now justify this selection.

Our approach is to view Web server architecture as a dependency graph, in which the nodes represent processes, services, users, and user state. An edge (a, b) denotes b 's dependence on a , meaning an attacker's ability to compromise a implies an ability to compromise b . The crucial design decision is thus how to establish dependencies between the more abstract notions of services, users and user states, and the more concrete notion of a process.

Let the set S represent a Web server's constituent services, and assume each service accesses a private pool of data. (Two application-level services that share data would thus be modelled by a single "service".) A set of users U interacts with these services, and the interaction between user u_j and service s_i involves a piece of state $t_{i,j}$. If an attacker can compromise a service s_i , he can compromise state $t_{i,j}$ for all j ; thus $(s_i, t_{i,j})$ is a dependency for all j . Compromising state also compromises the corresponding user, so $(t_{i,j}, u_j)$ is also a dependency.

Let $P = \{p_1, \dots, p_k\}$ be a Web server's pool of processes. The design decision of how to allocate processes reduces to where the nodes in P belong on the dependency graph. In the Apache architecture [3], each process p_i in the process pool can perform the role of any service s_j . Thus, dependencies (p_i, s_j) exist for all j . For Flash [3], each process in P is associated with a particular service: for each p_i , there exists s_j such that (p_i, s_j) is a dependency. The size of the process pool P is determined by the number of concurrent active HTTP sessions; each process p_i serves only one of these connections. Java-based systems like the Haboob Server [44] employ only one process; thus $P = \{p_1\}$, and dependencies (p_1, s_j) exist for all j .

Figures 2(a)-(c) depict graphs of Apache, Flash and Haboob hosting two services for two remote users. Assuming that the "dependence" relationship is transitive, and that an adversary can compromise p_1 , the shaded nodes in the graph show all other vulnerable entities.

This picture assumes that the process of p_1 is equally vulnerable in the different architectures and that all architectures succeed equally in isolating different processes from each other. Neither of these assumptions is entirely true, and we will return to these issues in Section 6.2. What is clear from these graphs is that in the case of Flash, a compromise of p_1 does not affect states $t_{2,1}$ and $t_{2,2}$. For example, an attacker who gained access to u_1 's search history $(t_{1,i})$ cannot access the contents of his inbox $(t_{2,i})$.

A more strict isolation strategy is shown in Figure 2(d). The architecture assigns a process p_i to each user u_i . If the attacker is a user u_i , he should only be able to compro-

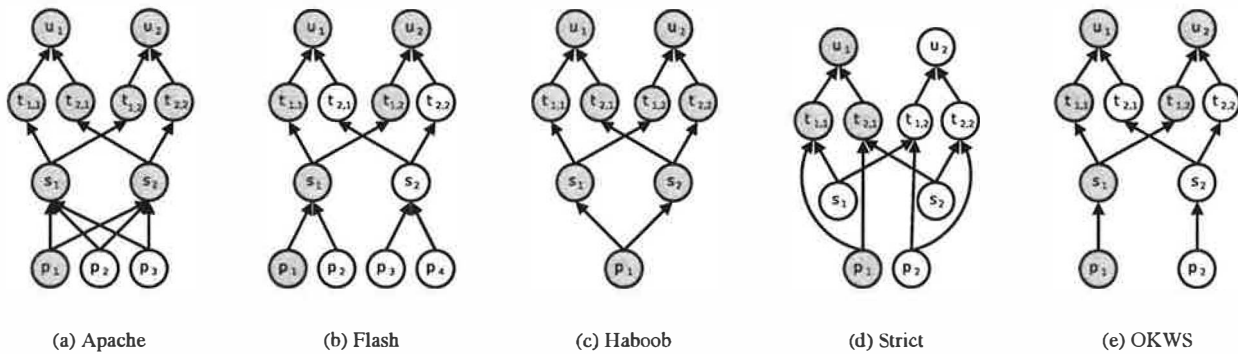


Figure 2: Dependency graphs for various Web server architectures.

mise his own process p_i , and will not have access to state belonging to other users u_j . The problem with this approach is that it does not scale well. A Web server would either need to fork a new process p_i for each incoming HTTP request or would have a large pool of mostly idle processes, one for each currently active user (of which there might be tens of thousands).

OKWS does not implement the strict isolation strategy but instead associates a single process with each individual service, shown in Figure 2(e). As a result OKWS achieves the same isolation properties as Flash but with a process pool whose size is independent of the number of concurrent HTTP connections.

4 Implementation

OKWS is a portable, event-based system, written in C++ with the SFS toolkit [18]. It has been successfully tested on Linux and FreeBSD. In OKWS, the different helper processes and site-specific services shown in Figure 1 communicate among themselves with SFS's implementation of Sun RPC [32]; they communicate with external Web clients via HTTP. Unlike other event-based servers [23, 44, 47], OKWS exposes the event architecture to Web developers.

To use OKWS, an administrator installs the helper binaries (*okld*, *okd*, *pubd* and *oklogd*) to a standard directory such as `/usr/local/sbin`, and installs the site-specific services to a runtime jail directory, such as `/var/okws/run`. The administrator should allocate two new UID/GID pairs for *okd* and *oklogd* and should also reserve a contiguous user and group ID space for “anonymous” services. Finally, administrators can tweak the master configuration file, `/etc/okws.config`. Table 1 summarizes the runtime configuration of OKWS.

4.1 okld

The root process in the OKWS system is *okld*—the launcher daemon. This process normally runs as super-user but can be run as a non-privileged user for testing or in other cases when the Web server need not bind to a privileged TCP port. When *okld* starts up, it reads the configuration file `/etc/okws.config` to determine the locations of the OKWS helper processes, the anonymous user ID range, which directories to use as jail directories, and which services to launch. Next, *okld* launches the logging daemon (*oklogd*) and the demultiplexing daemon (*okd*), and *chroots* into its runtime jail directory. It then launches all site-specific Web services. The steps for launching a single service are:

1. *okld* requests a new Unix socket connection from *oklogd*.
2. *okld* opens 2 socket pairs; one for HTTP connection forwarding, and one for RPC control messages.
3. *okld* calls *fork*.
4. In the child address space, *okld* picks a fresh UID/GID pair ($x.x$), sets the new process's group list to $\{x\}$ and its UID to x . It then changes directories into `/cores/x`.
5. Still in the child address space, *okld* calls *execve*, launching the Web service. The new Web service process inherits three file descriptors: one for receiving forwarded HTTP connections, one for receiving RPC control messages, and one for RPC-based request logging. Some configuration parameters in `/etc/okws.config` are relevant to child services, and *okld* passes these to new children via the command line.

process	chroot jail	run directory	uid	gid
<i>okld</i>	<i>/var/okws/run</i>	<i>/</i>	root	wheel
<i>pubd</i>	<i>/var/okws/htdocs</i>	<i>/</i>	www	www
<i>oklogd</i>	<i>/var/okws/log</i>	<i>/</i>	oklogd	oklogd
<i>okd</i>	<i>/var/okws/run</i>	<i>/</i>	okd	okd
<i>svc₁</i>	<i>/var/okws/run</i>	<i>/cores/51001</i>	51001	51001
<i>svc₂</i>	<i>/var/okws/run</i>	<i>/cores/51002</i>	51002	51002
<i>svc₃</i>	<i>/var/okws/run</i>	<i>/cores/51003</i>	51003	51003

Table 1: An example configuration of OKWS. The entries in the “run directory” column are relative to “chroot jails”.

6. In the parent address space, *okld* sends the server side of the sockets opened in Step 2 to *okd*.

Upon a service’s first launch, *okld* assigns it a group and user ID chosen arbitrarily from the given range (e.g., 51001-51080). The service gets those same user and group IDs in subsequent launches. It is important that no two services share a UID or GID, and *okld* ensures this invariant. The service executables themselves are owned by root, belong to the group with the anonymous GID *x* chosen in Step 4 and are set to mode 0410.

These settings allow *okld* to call `execve` after `setuid` but disallow a service process from changing the mode of its corresponding binary. *okld* changes the ownerships and permissions of service executables at launch if they are not appropriately set. The directory used in Step 4 is the only one in the jailed file system to which the child service can write. If such a directory does not exist or has the wrong ownership or permissions, *okld* creates and configures it accordingly.

okld catches `SIGCHLD` when services die. Upon receiving a non-zero exit status, *okld* changes the owner and mode of any core files left behind, rendering them inaccessible to other OKWS processes. If a service exits uncleanly too many times in a given interval, *okld* will mark it broken and refuse to restart it. Otherwise, *okld* restarts dead services following the steps enumerated above.

4.2 okd

The *okd* process accepts incoming HTTP requests and demultiplexes them based on the “Request-URI” in their first lines. For example, the HTTP/1.1 standard [11] defines the first line of a GET request as:

```
GET /<abs_path>?<query> HTTP/1.1
```

Upon receiving such a request, *okd* looks up a Web service corresponding to *abs_path* in its dispatch table. If successful, *okd* forwards the remote client’s file descriptor to the requested service. If the lookup is successful but the service is marked “broken,” *okd* sends an HTTP 500 error to the remote client. If the request did not match

a known service, *okd* returns an HTTP 404 error. In typical settings, a small and fixed number of these services are available—on the order of 10. The set of available services is fixed once *okd* reads its configuration file at launch time.

Upon startup, *okd* reads the OKWS configuration file (`/etc/okws.config`) to construct its dispatch table. It inherits two file descriptors from *okld*: one for logging, and one for RPC control messages. *okd* then listens on the RPC channel for *okld* to send it the server side of the child services’ HTTP and RPC connections (see Section 4.1, Step 6). *okd* receives one such pair for each service launched. The HTTP connection is the sink to which *okd* sends incoming HTTP requests from external clients after successful demultiplexing. Note that *okd* needs access to *oklogd* to log Error 404 and Error 500 messages.

okd also plays a role as a control message router for the child services. In addition to listening for HTTP connections on port 80, *okd* also listens for internal requests from an administration client. It services the two RPC calls: `REPUBLIC` and `RELAUNCH`. A site maintainer should call the former to “activate” any changes she makes to HTML templates (see Section 4.4 for more details). Upon receiving a `REPUBLIC` RPC, *okd* triggers a simple update protocol that propagates updated templates.

A site maintainer should issue a `RELAUNCH` RPC after updating a service’s binary. Upon receiving a `RELAUNCH` RPC, *okd* simply sends an EOF to the relevant service on its control socket. When a Web service receives such an EOF, it finishes responding to all pending HTTP requests, flushes its logs, and then exits cleanly. The launcher daemon, *okld*, then catches the corresponding `SIGCHLD` and restarts the service.

4.3 oklogd

All services, along with *okd*, log their access and error activity to local files via *oklogd*—the logger daemon. Because these processes lack the privileges to write to the same log file directly, they instead send log updates over a local Unix domain socket. To reduce the total number of messages, services send log updates in batches. Services flush their log buffers as they become full and at regularly-scheduled intervals.

For security, *oklogd* runs as an unprivileged user in its own *chroot* environment. Thus, a compromised *okd* or Web service cannot maliciously overwrite or truncate log files; it would only have the ability to fill them with “noise.”

4.4 *pubd*

Dynamic Web pages often contain large sections of static HTML code. In OKWS, such static blocks are called HTML “templates”; they are stored as regular files, can be shared by multiple services and can include each other in a manner similar to Server Side Includes [4].

OKWS services do not read templates directly from the file system. Rather, upon startup, the publishing daemon (*pubd*) parses and caches all required templates. It then ships parsed representations of the templates over RPC to other processes that require them. *pubd* runs as an unprivileged user, relegated to a jail directory that contains only public HTML templates. As a security precaution, *pubd* never updates the files it serves, and administrators should set its entire *chrooted* directory tree read-only (perhaps, on those platforms that support it, by mounting a read-only *nullfs*).

5 OKWS In Practice

Though its design is motivated by security goals, OKWS provides developers with a convenient and powerful toolkit. Our experience suggests that OKWS is suitable for building and maintaining large commercial systems.

5.1 Web Services

A Web developer creates a new Web service as follows:

1. Extends two OKWS generic classes: one that corresponds to a long-lived service, and one that corresponds to an individual HTTP request. Implements the *init* method of the former and the *process* method of the latter.
2. Runs the source file through OKWS’s preprocessor, which outputs C++ code.
3. Compiles this C++ code into an executable, and installs it in OKWS’s service jail.
4. Adds the new service to */etc/okws.config*.
5. Restarts OKWS to launch.

The resulting Web service is a single-threaded, event-driven process.

The OKWS core libraries handle the mundane mechanics of a service’s life cycle and its connections to OKWS helper processes. At the initialization stage, a Web service establishes persistent connections to all needed databases. The connections last the lifetime of the service and are automatically reopened in the case of

abnormal termination. Also at initialization, a Web service obtains static HTML templates and local configuration parameters from *pubd*. These data stay in memory until a message from *okd* over the RPC control channel signals that the Web service should refetch. In implementing the *init* method, the Web developer need only specify which database connections, templates and configuration files he requires.

The *process* method specifies the actions required for incoming HTTP requests. In formulating replies, a Web service typically accesses cached soft-state (such as user session information), database-resident hard state (such as inbox contents), HTML templates, and configuration parameters. Because a Web service is implemented as a single-threaded process, it does not require synchronization mechanisms when accessing these data sources. Its accesses to a database on behalf of all users are pipelined through a single asynchronous RPC channel. Similarly, its accesses to cached data are guaranteed to be atomic and can be achieved with simple lightweight data structures, without locking. By comparison, other popular Web servers require some combination of *mmap*’ed files, spin-locks, IPC synchronization, and database connection pooling to achieve similar results.

At present, OKWS requires Web developers to program in C++, using the same SFS event library that undergirds all OKWS helper processes and core libraries. To simplify memory management, OKWS exposes SFS’s reference-counted garbage collection scheme and high-level string library to the Web programmer. OKWS also provides a C++ preprocessor that allows for Perl-style “heredocs” and simplified template inclusion. Figure 3 demonstrates these facilities.

5.2 Asynchronous Database Proxies

OKWS provides Web developers with a generic library for translating between asynchronous RPC and any given blocking client library, in a manner similar to Flash’s helper processes [23], and “manual calling automatic” in [1]. OKWS users can thus simply implement *database proxies*: asynchronous RPC front-ends to standard databases, such as MySQL [21] or Berkeley DB [29]. Our libraries provide the illusion of a standard asynchronous RPC dispatch routine. Internally, these proxies are multi-threaded and can block; the library handles synchronization and scheduling.

Database proxies employ a small and static number of worker threads and do not expand their thread pool. The intent here is simply to overlap requests to the underlying data source so that it might overlap its disk accesses and

```

void my_srvc_t::process ()
{
    str color = param["color"];
    /*
    print (resp) <<EOF;
<html>
<head>
<title>${param["title"]}</title>
</head>
EOF
    include (resp, "/body.html",
            { COLOR => ${color}});
    o*/
    output (resp);
}

```

Figure 3: Fragment of a Web service programmed in OKWS. The remote client supplies the title and color of the page via standard CGI-style parameter passing. The runtime templating system substitutes the user's choice of color for the token `COLOR` in the template `/body.html`. The variable `my_srvc_t::resp` represents a buffer that collects the body of the HTTP response and then is flushed to the client via `output()`. With the `FilterCGI` flag set, OKWS filters all dangerous metacharacters from the `param` associative array.

benefit from disk arm scheduling.

Database proxies ought to run on the database machines themselves. Such a configuration allows the site administrator to “lock down” a socket-based database server, so that only local processes can execute arbitrary database commands. All other machines in the cluster—such as the Web server machines—only see the structured, and thus restricted, RPC interface exposed by the database proxy.

Finally, database proxies employ a simple mechanism for authenticating Web services. After a Web service connects to a database proxy, it supplies a 20-byte authentication token in a login message. The database proxy then grants the Web service permission to access a set of RPCs based on the supplied authentication token.

To facilitate development of OKWS database proxies, we wrapped MySQL's standard C library in an interface more suitable for use with SFS's libraries. We model our MySQL interface after the popular Perl DBI interface [24] and likewise transparently support both parsed and prepared SQL styles. Figure 4 shows a simple database proxy built with this library.

5.3 Real-World Experience

The author and two other programmers built a commercial Web site using the OKWS system in six months [22]. We were assisted by two designers who knew little C++ but made effective use of the HTML templating system.

The application is Internet dating, and the site features a typical suite of services, including local matching, global matching, messaging, profile maintenance, site statistics, and picture browsing. Almost a million users have established accounts on the site, and at peak times, thousands of users maintain active sessions. Our current implementation uses 34 Web services and 12 database proxies.

We have found the system to be usable, stable and well-performing. In the absence of database bottlenecks or latency from serving advertisements, OKWS feels very responsive to the end user. Even those pages that require iterative computations—like match computations—load instantaneously.

Our Web cluster currently consists of four load balanced OKWS Web server machines, two read-only cache servers, and two read-write database servers, all with dual Pentium 4 processors. We use multiple OKWS machines only for redundancy; one machine can handle peak loads (about 200 requests per second) at about 7% CPU utilization, even as it *gzips* most responses. A previous incarnation of this Web site required six ModPerl/Apache servers [20] to accommodate less traffic. It ultimately was abandoned due to insufficient software tools and prohibitive hardware and hosting expenses [30].

5.4 Separating Static From Dynamic

OKWS relies on other machines running standard Web servers to distribute static content. This means that all pages generated by OKWS should have only absolute links to external static content (such as images and style sheets), and OKWS has no reason to support keep-alive connections [11]. The servers that host static content for OKWS, however, can enable HTTP keep-alive as usual.

We note that serving static and dynamic content from different machines is already a common technique for performance reasons; administrators choose different hardware and software configurations for the two types of workloads. Moreover, static content service does not require access to sensitive site data and can therefore happen outside of a firewalled cluster, or perhaps at a different hosting facility altogether. Indeed, some sites push static content out to external distribution networks such as Akamai [2].

In our commercial deployment, we host a cluster of OKWS and database machines at a local colocation facility; we require hands-on hardware access and a network configured for our application. We serve static content from leased, dedicated servers at a remote facility where bandwidth is significantly cheaper.

```

struct user_xdr_t {
    string name<30>;
    int age;
};

// can only occur at initialization time
q = mysql->prepare (
    "SELECT age,name FROM tab WHERE id=?");

id = 1; // get ID from client
user_xdr_t u;
stmt = q->execute (id); // might block!
stmt->fetch (&u.age, &u.name);
reply (u);

```

Figure 4: Example of database proxy code with MySQL wrapper library. In this case, the Web developer is loading SQL results directly into an RPC XDR structure.

6 Security Discussion

In this section we discuss OKWS's security benefits and shortcomings.

6.1 Security Benefits

(1) *The Local Filesystem.* An OKWS service has almost no access to the file system when execution reaches custom code. If compromised, a service has write access to its coredump directory and can read from OKWS shared libraries. Otherwise, it cannot access `setuid` executables, the binaries of other OKWS services, or core dumps left behind by crashed OKWS processes. It cannot overwrite HTTP logs or HTML templates. Other OKWS services such as *oklogd* and *pubd* have more privileges, enabling them to write to and read from the file system, respectively. However, as OKWS matures, these helpers should not present security risks since they do not run site-specific code.

(2) *Other Operating System Privileges.* Because OKWS runs logically separate processes under different user IDs, compromised processes (with the exception of *okld*) do not have the ability to *kill* or *ptrace* other running processes. Similarly, no process save for *okld* can bind to privileged ports.

(3) *Database Access.* As described, all database access in OKWS is achieved through RPC channels, using independent authentication mechanisms. As a result, an attacker who gains control of an OKWS web service can only interact with the database in a manner specified by the RPC protocol declaration; he does not have generic SQL client access. Note that this is a stronger restriction than simple database permission

systems alone can guarantee. For instance, on PHP systems, a particular service might only have SELECT permissions to a database's USERS table. But with control of the PHP server, an attacker could still issue commands like SELECT * FROM USERS. With OKWS, if the RPC protocol restricts access to row-wise queries and the keyspace of the table is sparse, the attacker has significantly more difficulty "mining" the database.¹

OKWS's separation of code and privileges further limits its attacks. If a particular service is compromised, it can establish a new connection to a remote RPC database proxy; however, because the service has no access to source code, binaries, or *ptraces* of other services, it knows no authentication tokens aside from its own.

Finally, OKWS database libraries provide runtime checks to ensure that SQL queries can be prepared only when a proxy starts up, and that all parameters passed to queries are appropriately escaped. This check insulates sloppy programmers from the "SQL injection" attacks mentioned in Section 2.2. We expect future versions of OKWS to enforce the same invariants at compile time.

(4) *Process Isolation and Privilege Separation.* OKWS is careful to separate the traditionally "buggy" aspects of Web servers from the most sensitive areas of the system. In particular, those processes that do the majority of HTTP parsing (the OKWS services) have the fewest privileges. By the same logic, *okld*, which runs as superuser, does no message parsing; it responds only to signals. For the other helper processes, we believe the RPC communication channels to be less error-prone than standard HTTP messaging and unlikely to allow intruders to traverse process boundaries.

Process isolation also limits the scope of those DoS attacks that exploit bugs in site-specific logic. Since the operating system sets per-process limits on resources such as file descriptors and memory, DoS vulnerabilities should not spread across process boundaries. We could make stronger DoS guarantees by adapting "defensive programming" techniques [26]. Qie *et al.* suggest compiling rate-control mechanisms into network services, for dynamic prevention of DoS attacks. Their system is applicable within OKWS's architecture, which relegates each service to a single address space. The same cannot be said for those systems that spread equivalent functionality across multiple address spaces.

6.2 Security Shortcomings

The current implementation of OKWS supports only C++ for service development. OKWS programmers

```

<html><head><title>Test Result</title></head>
<body>
<?
$db = mysql_pconnect("okdb.lcs.mit.edu");
mysql_select_db("testdb", $db);
$id = $_HTTP_GET_VARS["id"];
$qry = "SELECT x,y FROM tab WHERE x=$id";
$result = mysql_query("$qry", $db);
$myrow = mysql_fetch_row($result);
print("QRY $id $myrow[0] $myrow[1]\n");
?>
</body>
</html>

```

Figure 5: PHP version of the null service

should use the provided “safe” strings classes when generating HTML output, and they should use only auto-generated RPC stubs for network communication; however, OKWS does not prohibit programmers from using unsafe programming techniques and can therefore be made more susceptible to buffer overruns and stack-smashing attacks. Future versions of OKWS might make these attacks less likely by supporting higher-level programming languages such as Python or Perl.

Another shortcoming of OKWS is that an adversary who compromises an OKWS service can gain access to in-memory state belonging to other users. Developers might protect against this attack by encrypting cache entries with a private key stored in an HTTP cookie on the client’s machine. Encryption cannot protect against an adversary who can compromise and passively monitor a Web server.

Finally, independent aspects of the system might be vulnerable due to a common bug in the core libraries.

7 Performance Evaluation

In designing OKWS we decided to limit its process pool to a small and fixed size. In our evaluation, we tested the hypothesis that this decision has a positive impact on performance, examining OKWS’s performance as a function of the number of active service processes. We also present and test the claim that OKWS can achieve high throughputs relative to other Web servers because of its smaller process pool and its specialization for dynamic content.

7.1 Testing Methodology

Performance testing on Web servers usually involves the SPECweb99 benchmark [31], but this benchmark is not well-suited for dynamic Web servers that disable Keep-Alive connections and redirect to other machines for static content. We therefore devised a simple benchmark

that better models serving dynamic content in real-world deployments, which we call the *null service benchmark*. For each of the platforms tested, we implemented a *null service*, which takes an integer input from a client, makes a database SELECT on the basis of that input, and returns the result in a short HTML response (see Figure 5). Test clients make one request per connection: they connect to the server, supply a randomly chosen query, receive the server’s response, and then disconnect.

7.2 Experimental Setup

All Web servers tested use a large database table filled with sequential integer keys and their 20-byte SHA-1 hashes [12]. We constrained our client to query only the first 1,000,000 rows of this table, so that the database could store the entire dataset in memory. Our database was MySQL version 4.0.16.

All experiments used four FreeBSD 4.8 machines. The Web server and database machines were uniprocessor 2.4GHz and 2.6GHz Pentium 4s respectively, each with 1GB of RAM. Our two client machines ran Dual 3.0GHz Pentium 4s with 2GB of RAM. All machines were connected via fast Ethernet, and there was no network congestion during our experiments. Ping times between the clients and the Web server measured around 250 μ s, and ping times between the Web server and database machine measured about 150 μ s.

We implemented our test client using the OKWS libraries and the SFS toolkit. There was no resource strain on the client machines during our tests.

7.3 OKWS Process Pool Tests

We experimentally validated OKWS’s frugal process allocation strategy by showing that the alternative—running many processes per service—performs worse. We thus configured OKWS to run a single service as a variable number of processes, and collected throughput measurements (in requests per second) over the different configurations. The test client was configured to simulate either 500, 1,000 or 2,000 concurrent remote clients in the different runs of the experiment.

Figure 6 summarizes the results of this experiment as the number of processes varied between 1 and 450. We attribute the general decline in performance to increased context-switching, as shown in Figure 7. In the single-process configuration, the operating system must switch between the null service and *okd*, the demultiplexing daemon. In this configuration, higher client concurrency implies fewer switches, since both *okd* and the null service have more outstanding requests to service before calling

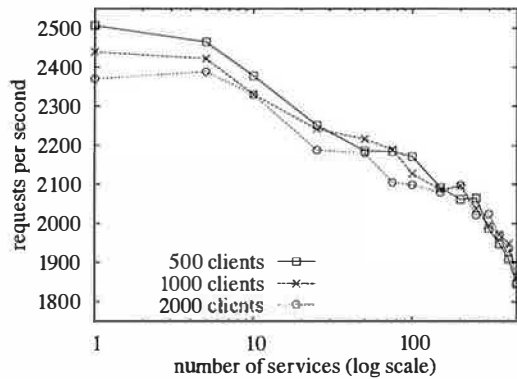


Figure 6: Throughputs achieved in the process pool test

sleep. This effect quickly disappears as the server distributes requests over more processes. As their numbers grow, each process has, on average, fewer requests to service per unit of time, and therefore calls *sleep* sooner within its CPU slice.

The process pool test supports our hypothesis that a Web server will consume more *computational* resources as its process pool grows. Although the experiments completed without putting *memory* pressure on the operating system, memory is more scarce in real deployments. The null service requires about 1.5MB of core memory, but our experience shows real OKWS service processes have memory footprints of at least 4MB, and hence we expect memory to limit server pool size. Moreover, in real deployments there is less memory to waste on code text, since in-memory caches on the Web services are crucial to good site performance and should be allowed to grow as big as possible.

7.4 Web Server Comparison

The other Web servers mentioned in Section 3.3—Haboob, Flash and Apache—are primarily intended for serving static Web pages. Because we have designed and tuned OKWS for an entirely dynamic workload, we hypothesize that when servicing such workloads, it performs better than its more general-purpose peers. Our experiments in this section test this hypothesis.

Haboob is Java-based, and we compiled and ran it with FreeBSD's native JDK, version 1.3. We tested Flash v0.1a, built with `FD_SETSIZE` set high so that Flash reported an ability to service 5116 simultaneous connections. Also tested was Apache version 2.0.47 compiled with multi-threading support and running PHP version 4.3.3 as a dynamic shared object. We configured Apache to handle up to 2000 concurrent connections. We ran OKWS in its standard configuration, with a one-to-one

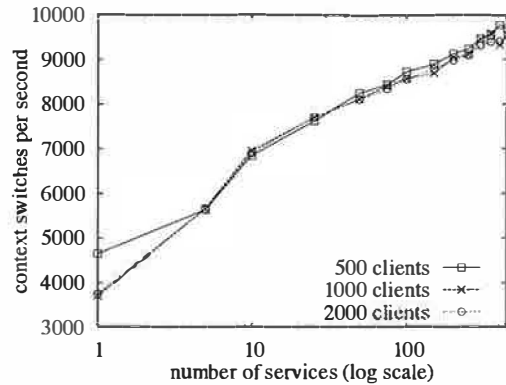


Figure 7: Context switching in the process pool test

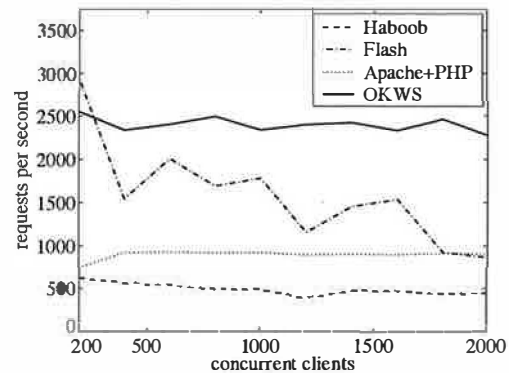


Figure 8: Throughputs for the single-service test

correspondence between processes and services.

We enabled HTTP access logging on all systems with the exception of Haboob, which does not support it. All systems used persistent database connections.

7.4.1 Single-Service Workload

In the single-service workload, clients with negligible latency request a dynamically generated response from the null service. This test entails the minimal number of service processes for OKWS and Flash and therefore should allow them to exhibit maximal throughput. By contrast, Apache and Haboob's process pools do not vary in size with the number of available services. We examined the throughput (Figure 8) and responsiveness (Figure 9) of the four systems as client concurrency increased. Figure 10 shows the cumulative distribution of client latencies when 1,600 were active concurrently.

Of the four Web servers tested, Haboob spent the most CPU time in user mode and performed the slowest. A likely cause is the sluggishness of Java 1.3's memory management.

When servicing a small number of concurrent clients, the Flash system outperforms the others; however, its per-

formance does not scale well. We attribute this degradation to Flash's CGI model: because custom-written Flash helper processes have only one thread of control, each instantiation of a helper process can handle only one external client. Thus, Flash requires a separate helper process for each external client served. At high concurrency levels, we noted a large number of running processes (on the order of 2000) and general resource starvation. Flash also puts additional strain on the database, demanding one active connection per helper—thousands in total. A database pooling system might mitigate this negative performance impact. Flash's results were noisy in general, and we can best explain the observed non-monotonicity as inconsistent operating system (and database) behavior under heavy strain.

Apache achieves 37% of OKWS's throughput on average. Its process pool is bigger and hence requires more frequent context switching. When servicing 1,000 concurrent clients, Apache runs around 450 processes, and context switches about 7500 times a second. We suspect that Apache starts queuing requests unfairly above 1,000 concurrent connections, as suggested by the plateau in Figure 9 and the long tail in Figure 10.

In our configuration, PHP makes frequent calls to the *sigprocmask* system call to serialize database accesses among kernel threads within a process. In addition, Apache makes frequent (and unnecessary) file system accesses, which though serviced from the buffer cache still entail system call overhead. OKWS can achieve faster performance because of a smaller process pool and fewer system calls.

7.4.2 Many-Service Workload

In attempt to model a more realistic workload, we investigated Web servers running more services, serving more data, as experienced by clients over the WAN. We modified our null services to send out an additional 3000 bytes of text with every reply (larger responses would have saturated the Web server's access link in some cases). We made 10 uniquely-named copies of the new null service, convincing the Web servers that they were serving 10 distinct services. Finally, our clients were modified to pause an average of 75 ms between establishing a connection and sending an HTTP request. We ran the experiment from 200 to 2000 simultaneous clients, and observed a graph similar in shape to Figure 8.

Achieved throughputs are shown in Table 2 and are compared to the results observed in the single-service workload. Haboob's performance degrades most notably, probably because the many-service workload demands more memory allocations. Flash's throughput decreases

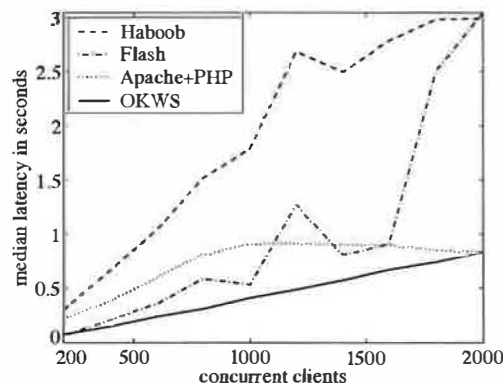


Figure 9: Median Latencies

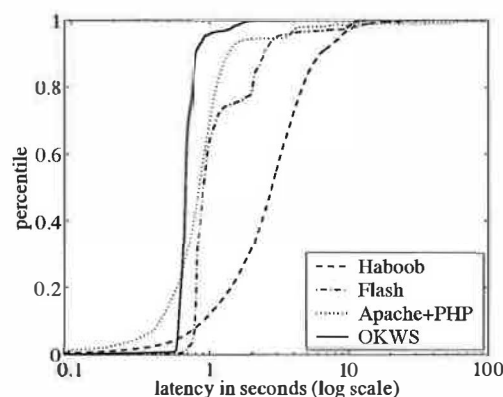


Figure 10: Client latencies for 1,600 concurrent clients

by 23%. We observed that for this workload, Flash requires even more service processes, and at times over 2,500 were running. When we switched from the single-service to the many-service configuration, the number of active OKWS services increased from 1 to 10. The results from Figure 6 show this change has little impact on throughput. We can better explain OKWS's diminished performance by arguing that larger HTTP responses result in more data shuffling in user mode and more pressure on the networking stack in kernel mode. The same explanation applies for Apache, which experienced a similar performance degradation.

8 Related work

Apache's [3] many configuration options and modules allow Web programmers to extend its functionality with a variety of different programming languages. However, neither 1.3.x's multi-process architecture nor 2.0.x's multi-threaded architecture is conducive to process isolation. Also, its extensibility and mushrooming code base make its security properties difficult to reason about.

	Haboob	Apache	Flash	OKWS
1 Service	490	895	1,590	2,401
10 Services	225	760	1,232	2,089
Change	-54.0%	-15.1%	-22.5%	-13.0%

Table 2: Average throughputs in connections per second

Highly-optimized event-based Web servers such as Flash [23] and Zeus [47] have eclipsed Apache in terms of performance. While Flash in particular has a history of outstanding performance serving static content, our performance studies here indicate that its architecture is less suitable for dynamic content. In terms of process isolation, one could most likely implement a similar separation of privileges in Flash as we have done with OKWS.

FastCGI [10] is a standard for implementing long-lived CGI-like helper processes. It allows separation of functionality along process boundaries but neither articulates a specific security policy nor specifies the mechanics for maintaining process isolation in the face of partial server compromise. Also, FastCGI requires the leader process to relay messages between the Web service and the remote client. OKWS passes file descriptors to avoid the overhead associated with FastCGI's relay technique.

The Haboob server studied here is one of many possible applications built on SEDA, an architecture for event-based network servers. In particular, SEDA uses serial event queues to enforce fairness and graceful degradation under heavy load. Larger systems such as Ninja [33] build on SEDA's infrastructure to create clusters of Web servers with the same appealing properties.

Other work has used the SFS toolkit to build static Web Servers and Web proxies [46]. Though the current OKWS architecture is well-suited for SMP machines, the adoption of *libasync-mp* would allow for finer-grained sharing of a Web workload across many CPUs.

OKWS uses events but the same results are possible with an appropriate threads library. An expansive body of literature argues the merits of one scheme over the other, and most recently, Capriccio's authors [34] argue that threads can achieve the same performance as events in the context of Web servers, while providing programmers with a more intuitive interface. Other recent work suggests that threads and events can coexist [1]. Such techniques, if applied to OKWS, would simplify stack management for Web developers.

In addition to the PHP [25] scripting language investigated here, many other Web development environments are in widespread use. Zope [48], a Python-based platform, has gained popularity due to its modularity and support for remote collaboration. CSE [13] allows devel-

opers to write Web services in C++ and uses some of the same sandboxing schemes we use here to achieve fault isolation. In more commercial settings, Java-based systems often favor thin Web servers, pushing more critical tasks to application servers such as JBoss [15] and IBM WebSphere [14]. Such systems limit a Web server's access to underlying databases in much the same way as OKWS's database proxies. Most Java systems, however, package all aspects of a system in one address space with many threads; our model for isolation would not extend to such a setting. Furthermore, our experimental results indicate significant performance advantages of compiled C++ code over Java systems.

Other work has proposed changes to underlying operating systems to make Web servers fast and more secure. The Exokernel operating system [16] allows its Cheetah Web server to directly access the TCP/IP stack, in order to reduce buffer copies allow for more effective caching. The Denali isolation kernel [45] can isolate Web services by running them on separate virtual machines.

9 Summary and Future Work

OKWS is a toolkit for serving dynamic Web content, and its architecture fits naturally into a compelling security model. The system's separation of processes provides reasonable assurances that vulnerabilities in one aspect of the system do not metastasize. The performance results we have seen are encouraging: OKWS derives significant speedups from a small and fixed process pool, lightweight synchronization mechanisms, and avoidance of unnecessary system calls. In the future, we plan to experiment with high-level language support and better resilience to DoS attacks. Independent of future improvements, OKWS is stable and practical, and we have used it to develop a popular commercial product.

Acknowledgments

I am indebted to David Mazières for his help throughout the project, and to my advisor Frans Kaashoek for help in preparing this paper. Michael Walfish significantly improved this paper's writing and presentation. My shepherd Eddie Kohler suggested many important improvements, Robert Morris assisted in debugging, and the anonymous reviewers provided insightful comments. I thank the programmers, designers and others at OkCupid.com—Patrick Crosby, Jason Yung, Chris Coyne, Christian Rudder and Sam Yagan—for adopting and improving OKWS, and Sarah Friedberg for proof-reading. This research was supported in part by the DARPA Composable High Assurance Trusted Systems

program (BAA #01-24), under contract #N66001-01-1-8927.

Availability

OKWS is available under an open source license at www.okws.org.

References

- [1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proceedings of the 2002 USENIX*, Monterey, CA, June 2002. USENIX.
- [2] Akamai Technologies, Inc. <http://www.akamai.com>.
- [3] The Apache Software Foundation. <http://www.apache.org>.
- [4] Apache Tutorial: Introduction to Server Side Includes. <http://httpd.apache.org/docs/howto/ssi.html>.
- [5] Bugtraq ID 4606. SecurityFocus. <http://www.securityfocus.com/bid/4606/info/>.
- [6] Bugtraq ID 5993. SecurityFocus. <http://www.securityfocus.com/bid/5993/info/>.
- [7] Bugtraq ID 7255. SecurityFocus. <http://www.securityfocus.com/bid/7255/info/>.
- [8] Bugtraq ID 8138. SecurityFocus. <http://www.securityfocus.com/bid/8138/info/>.
- [9] CERT® Coordination Center. <http://www.cert.org>.
- [10] Open Market. Fastcgi. <http://www.fastcgi.com>.
- [11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol — HTTP/1.1*. Internet Network Working Group RFC 2616, 1999.
- [12] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.
- [13] T. Gchwind and B. A. Schmit. CSE — a C++ servlet environment for high-performance web applications. In *Proceedings of the FREEIX Track: 2003 USENIX Technical Conference*, San Antonio, TX, 2003. USENIX.
- [14] IBM corporation. IBM websphere application server. <http://www.ibm.com>.
- [15] JBoss Group. <http://www.jboss.org>.
- [16] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. ACM.
- [17] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003. ACM.
- [18] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX*. USENIX, June 2001.
- [19] Microsoft Corporation. IIS. <http://www.microsoft.com/windowsserver2003/iis/default.msp>.
- [20] mod_perl. <http://perl.apache.org>.
- [21] MySQL. <http://www.mysql.com>.
- [22] OkCupid.com. <http://www.okcupid.com>.
- [23] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX*, Monterey, CA, June 1999. USENIX.
- [24] PerlDBI. <http://dbi.perl.org>.
- [25] PHP: Hypertext processor. <http://www.php.net>.
- [26] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, October 2002. USENIX.
- [27] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, 1975.
- [28] SecurityFocus. <http://www.securityfocus.com>.
- [29] Sleepycat Software. <http://www.sleepycat.com>.
- [30] The SparkMatch service. Previously available at <http://www.thespark.com>.
- [31] Standard performance evaluation corporation. the specweb99 benchmark. <http://www.spec99.org/osg/web99/>.
- [32] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, August 1995.
- [33] J. R. van Berhen, E. A. Brewer, N. Borisova, M. C. an Matt Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 USENIX*, Monterey, CA, June 2002. USENIX.
- [34] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003. ACM.
- [35] Vulnerability CAN-2001-1246. SecurityFocus. <http://www.securityfocus.com/bid/2954/info/>.
- [36] Vulnerability CAN-2002-0656. SecurityFocus. <http://www.securityfocus.com/bid/5363/info/>.
- [37] Vulnerability CAN-2003-0132. SecurityFocus. <http://www.securityfocus.com/bid/7254/info/>.
- [38] Vulnerability CAN-2003-0253. <http://www.securityfocus.com/bid/8137/info/>.
- [39] Vulnerability CAN-2003-0542. SecurityFocus. <http://www.securityfocus.com/bid/8911/info/>.
- [40] Vulnerability CVE-2002-0061. SecurityFocus. <http://www.securityfocus.com/bid/4435/info/>.
- [41] Vulnerability Note VU117243. CERT. <http://www.kb.cert.org/vuls/id/910713>.
- [42] Vulnerability Note VU91073. CERT. <http://www.kb.cert.org/vuls/id/910713>.
- [43] Vulnerability Note VU979793. CERT. <http://www.kb.cert.org/vuls/id/979793>.
- [44] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001. ACM.
- [45] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, October 2002. USENIX.
- [46] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX*, San Antonio, TX, June 2003. USENIX.
- [47] Zeus Technology Limited. Zeus Web Server. <http://www.zeus.co.uk>.
- [48] The Zope Corporation. <http://www.zope.org>.

Notes

¹Similar security properties are possible with a standard Web server and a database that supports stored procedures, views, and roles.

REX: Secure, Extensible Remote Execution

Michael Kaminsky, Eric Peterson, Daniel B. Giffin,
Kevin Fu, David Mazières, M. Frans Kaashoek

*MIT Computer Science and Artificial Intelligence Laboratory,
NYU Department of Computer Science*

kaminsky@csail.mit.edu, ericp@csail.mit.edu, dbg@cs.nyu.edu,
fubob@csail.mit.edu, dm@cs.nyu.edu, kaashoek@csail.mit.edu

Abstract

The ubiquitous SSH package has demonstrated the importance of secure remote login and execution. As remote execution tools grow in popularity, users require new features and extensions, which are difficult to add to existing systems. REX is a remote execution utility with a novel architecture specifically designed for extensibility as well as security and transparent connection persistence in the face of network complexities such as NAT and dynamic IP addresses. To achieve extensibility, REX bases much of its functionality on a single new abstraction—*emulated file descriptor passing across machines*. This abstraction is powerful enough for users to extend REX's functionality in many ways without changing the core software or protocol.

REX addresses security in two ways. First, the implementation internally leverages file descriptor passing to split the server into several smaller programs, reducing both privileged and remotely exploitable code. Second, REX selectively delegates authority to processes running on remote machines that need to access other resources. The delegation mechanism lets users incrementally construct trust policies for remote machines. Finally, REX provides mechanisms for accessing servers without globally routable IP addresses, and for resuming sessions when a TCP connection aborts or an endpoint's IP address changes. Measurements of the system demonstrate that REX's architecture does not come at the cost of performance.

1 Introduction

Remote login and execution are network facilities that many people need for their day-to-day computing. The concept of remote login is simple: local input is fed to a program on a remote machine, and the program's output is sent back to the local terminal. In practice, however, modern remote login tools have become quite complex.

The popular SSH [38] program demonstrates that users expect features such as TCP port and X Window System forwarding, facilities for copying files back and forth, cryptographic user authentication, integration with network file systems, transfer of user credentials across machines, pseudo-terminals, and more. Many of these features require changes to the remote login protocol, for which developers add new message types.

Moreover, many users want other features that are not yet available: limitations on the amount of code subject to remote exploits, convenient trust management policies, transparent access to servers behind network address translation (NAT), and support for long-running remote login sessions when the client and server both change their IP addresses. The challenge in designing and building a remote execution tool is to address this diverse set of needs in a single, simple, extensible framework.

This paper introduces a new remote login and execution utility called REX, which has three main goals: extensibility, security, and transparent connection persistence despite NAT and dynamic IP addresses. The main contribution of REX is its architecture centered around *file descriptor passing*, both as an internal implementation technique and as an external interface highly amenable to extensions.

Extensibility. REX's approach to extensibility is for the core system and protocol to provide the simplest possible interface on which external utilities can implement advanced features like remote pseudo-terminal access, port forwarding, and authentication delegation. This interface consists principally of file descriptor passing: a process on one machine can effectively transfer a file descriptor to a process on another machine. In reality, REX emulates this operation by receiving the descriptor on one machine, passing a new socket to the recipient on the other machine, and subsequently relaying data back and forth between the descriptor and new socket over a cryptographically protected TCP connection. REX does

not care if a passed file descriptor is the master side of a pseudo-terminal, a connection from an X-windows client, a forwarded authentication agent connection, or some as-yet-unanticipated future extension.

Security. REX was designed from the ground up to minimize both the amount of code that runs with superuser privileges and the amount of code that deals directly with incoming network connections (which presents the greatest risk of being remotely exploitable). The REX server is split into two components: a small trusted program, *rex*, and a slightly larger, unprivileged, per-user program *proxy*. Remote clients can communicate only with *rex* until they prove that they are acting on behalf of an authorized user. *Proxy*, in turn, actually implements almost the entirety of what one would consider remote execution functionality. This separation of functions and privileges is possible because *rex* uses local file descriptor passing to hand off incoming connections to *proxy*.

The latest versions of OpenSSH [21] have moved in a similar direction by embracing privilege separation [24, 26]. The SSH protocol, however, was not designed to facilitate such an architecture, and the complexity of the implementation reflects this fact. For example, in one step, SSH must extract the memory heap from a process and essentially recreate it in another process's address space. Moreover, even the least privileged, "jailed," SSH processes still require the potentially dangerous ability to sign with the server's secret key.

A second security goal of REX is to provide precise delegation of authority to processes running on remote machines. Delegation of authority allows a remote process to access and authenticate itself to remote resources. However, a user might trust the remote machine less than the local one. To address this problem, REX can prompt users to authorize remote accesses on a case-by-case basis; by optionally instructing the agent to allow similar accesses in the future, users can build trust policies incrementally.

Transparent Connection Persistence. REX provides transparent connection persistence in the face of complex network configurations. The prevalence of network address translation (NAT) makes it hard to run globally accessible servers on many machines, while dynamically assigned IP addresses can disrupt long-running sessions. REX lets users transparently connect to remote login servers behind NAT boxes using either an externally addressable proxy server or DNS SRV records [12] (in conjunction with static TCP port mapping). REX's automatic connection resumption allows clients and servers to change IP addresses during the course of a connection.

We have built REX as part of the SFS [20] computing environment. REX currently offers modules that handle pseudo-terminal support, TCP port forwarding, X11

forwarding with cookie authentication, and Unix-domain socket forwarding. REX adds only two small pieces of privileged code to the system. One of these, the pseudo-terminal daemon, is only 400 lines of code and never touches an Internet socket; it is therefore unlikely to be remotely exploitable. The other, *rex*, is only 500 lines of code (not counting general-purpose crypto and Remote Procedure Call libraries). REX is in daily use, it runs on Unix, and the source code is freely available.

The rest of the paper is structured as follows. Section 2 describes REX's architecture, and Sections 3, 4, and 5 detail the main benefits of this architecture: extensibility, security, and transparency. Section 6 gives an evaluation of the implementation in terms of code size and performance. Finally we discuss related work, primarily regarding remote execution, and conclude.

2 Architecture

REX is designed to work with the Self-certifying File System (SFS), a secure, global network file system. SFS provides REX's user and server authentication facilities. REX also shares SFS's RPC compiler and library, which promote security by offering a concisely-specifiable communication interface between local and remote components, and by parsing messages with mechanically-generated code. Further, the use of local file descriptor passing allows REX to be broken into small functional units, minimizing the amount of privileged code.

The REX architecture offers extensibility through a communication abstraction that connects remote code (including arbitrary user programs) through the familiar interface of file descriptors. These pieces of code are called *modules*. REX groups file descriptors into *channels*, and channels into *sessions*. A *session* corresponds to all cryptographically protected communication over a single TCP connection between a REX client and a particular server. For each pair of communicating modules, there exists a *channel* within some session. Each channel can contain an arbitrary number of *file descriptor* pairs, over which modules may send data or more file descriptors.

Sections 2.1 and 2.2 provide background on user authentication and local and remote file descriptor passing. Sections 2.3 and 2.4 describe how REX establishes new sessions and how the channel abstraction is used to connect modules. Finally, this section concludes with a discussion of connection caching.

2.1 User Authentication in SFS

The key SFS subsystem that REX leverages is the user authentication infrastructure, which consists of two programs. The first is a per-user agent process, *sfsagent*,

which runs on the client machine. The agent stores a user's private key and signs authentication requests on his behalf. The second program is an authentication server, *sfsauthd*, which runs on the server machine. The authentication server verifies the signatures on authentication requests and then maps user public keys to local Unix accounts based on a database of registered SFS users.

2.2 File descriptor passing

File descriptors are numerical handles which name an opened file, socket, device, or other file-like resource. Most I/O in Unix is performed by reading from and writing to file descriptors. Unix also provides a facility for passing a file descriptor to another process through the *sendmsg* and *recvmsg* system calls on Unix-domain sockets [23].

REX uses local file descriptor passing between daemons, particularly on the server. This mechanism makes it easy to split functionality at a connection endpoint between a privileged and unprivileged process, typically by handing the connection from the privileged to the unprivileged process after some initialization phase. The use of local file descriptor passing as it relates to security is discussed further in Section 4.

REX also introduces the emulation of file descriptor passing between machines. This mechanism allows many extensions such as port forwarding and pseudo-terminal allocation to be implemented outside of the core system, thereby increasing extensibility. The use of file descriptor passing over the network is described in more detail in Section 3.

2.3 Sessions

Figures 1 and 2 show how REX establishes a session between a client machine (left) and a server (right). Boxes with a gray background are SFS programs that REX uses, while boxes with a white background are part of REX. Boxes with a filled upper-right corner are programs that run with superuser privileges. (The SFS agent is half-gray, half-white because even though it was part of the original SFS architecture, we extended it to support REX as described below.)

Setting up a REX session has two stages. In Stage I, the client establishes a secure, authenticated connection to the server. We call this initial connection the "master" REX session. In Stage II, the client creates new REX sessions, based on the master session, to run programs on the server.

2.3.1 Stage I

The user invokes the *rex* client¹ in order to start a new REX session. First, *rex* contacts the user's agent and asks it to establish a session to the desired server (Figure 1, Step 1). In Step 2, the *sfsagent* uses the server's public key to establish a secure connection to the *rex*d process running on the server.²

Mazières et al. [20] describe several mechanisms through which the client can obtain the server's key. By default, REX, like SSH, maintains a cache of server public keys that it has already seen. REX, however, avoids possible man-in-the-middle attacks when contacting a server for the first time by using the Secure Remote Password (SRP) protocol [35].

Next, the *sfsagent* authenticates its user to *rex*d (Step 3). The agent signs an authentication request, which it passes to the server through the secure connection. *Rex*d passes the authentication request to the authentication server, *sfsauthd*, which verifies the signature and identifies the user (maps the user's public key to a local account).

Once the user is authenticated, *rex*d, which runs with superuser privileges, spawns a new process called *proxy*, which runs with the privileges of the local user identified above (Step 4). *Rex*d uses file descriptor passing to hand the secure connection to *proxy*, which processes remote execution requests from the user (Step 5). The *sfsagent* maintains a connection to *proxy* in order to keep this master REX session alive; once the agent closes its connection to *proxy* (provided no other clients are still connected), *proxy* will exit and *rex*d will delete the master session. The master REX session is the basis for subsequent sessions between this user and server.

2.3.2 Stage II

To run a program on the server, the *rex* client notifies the user's *sfsagent* that it wants to create a new session to the given server (Figure 2, Step 1). *Sfsagent* looks up the corresponding master REX session and hands *rex* session keys for a new session to the same *proxy*. *Rex* then connects to *rex*d (Step 2). *Rex*d checks that *rex* indeed possesses appropriate keys, and if so hands the connection off to *proxy* through file descriptor passing (Step 3). Finally, *rex* asks *proxy* to spawn a program, say */bin/ls*, with a certain number of file descriptors (Step 4). The file descriptors inherited by */bin/ls* constitute a *channel*.

¹This paper will use REX (capital letters) to refer to the remote execution facility as a whole and *rex* (italicized lowercase) to refer to the client program that the user invokes to start a REX session.

²Since the SFS file server, authentication server, and *rex*d all listen on the same TCP port, connection setup by default also goes through an *sfsd* "meta-server." *Sfsd* demultiplexes incoming connections and hands them off to the appropriate daemon using file descriptor passing.

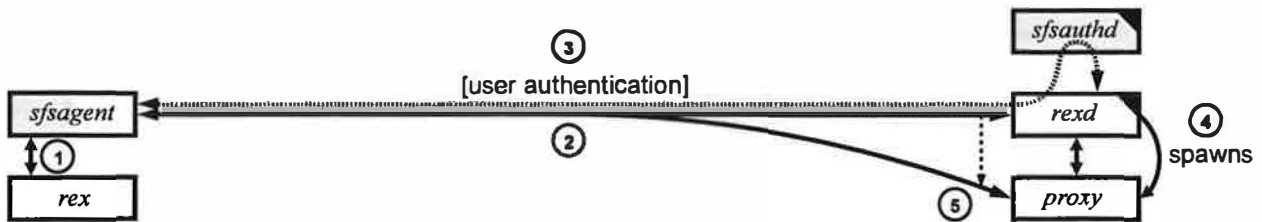


Figure 1: Setting up a REX session (Stage I)

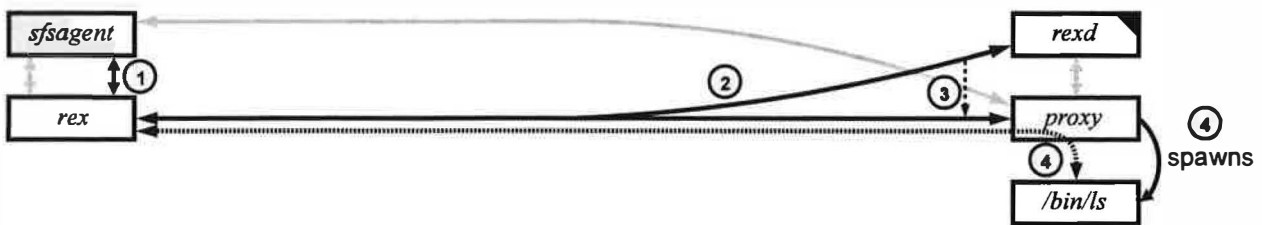


Figure 2: Setting up a REX session (Stage II). Gray lines represent connections that were established during Stage I.

Rex can relay data between the channel's file descriptors and a local module, or else connect the channel to its own standard input, output, and error.

2.4 Channels

The REX channel abstraction allows a pair of modules on different machines to communicate as if they were running on the same machine, connected by one or more Unix-domain sockets. When the client module writes data to a file descriptor, *rex* encapsulates that data as an RPC and sends it to *proxy*, which in turn unpacks the data and writes it to the appropriate file descriptor. The server module can then read the data on its corresponding file descriptor. *Proxy* similarly relays any data it reads back to *rex*.

The client creates channels through an RPC that specifies the name of the server module to run, a set of command line arguments and environment variables to set, and the number file descriptors the spawned module should inherit. (If fewer than three file descriptors are specified, standard input, standard output, and possibly standard error of the spawned process will be the same socket.) Depending on the channel, *rex* can either redirect I/O to a local module, or else relay data between the channel file descriptors and its own standard input, output, and error.

Channels are the mechanism through which REX emulates file descriptor passing over the network. When a module passes a file descriptor to *rex*, *rex* notifies *proxy* through an RPC. *Proxy* then creates a new Unix-domain

socket pair, passes one end to the local module, and allocates a new file descriptor number within the channel for the other end. Conversely, when a module passes a file descriptor to *proxy*, *proxy* allocates a new file descriptor number for it within the appropriate channel and notifies *rex*, which similarly passes one end of a new socket pair to the local module. As Section 3 demonstrates in detail, this emulated file descriptor passing is the foundation of REX's extensibility.

2.5 Connection caching

The REX protocol naturally lends itself to *connection caching* [6, 10]. Because *rex* uses the *sfsagent* to establish a master session with *rex*/*proxy* first, the *sfsagent* can remember (maintain) that connection and use it to set up subsequent REX sessions quickly. The initial REX connection to a remote machine is set up using public-key cryptography. Once this connection is established, REX uses symmetric cryptography to secure communication over the untrusted network. Subsequent REX connections to the same machine can bypass the public-key step and immediately begin encrypting the connection using symmetric cryptography.

For an interactive remote terminal session, the extra time required for the public-key cryptography might go unnoticed, but for batched remote execution that might involve tens or even hundreds of logins, the delay is observable. Connection caching offers an added benefit; if the user's agent was forwarded, that forwarding can remain in place even after the user logs out, allowing him

$$\begin{aligned}
\text{SessionKeySC}_i &= \text{HMAC-SHA-1}(\text{MasterSessionKeySC}, i) \\
\text{SessionKeyCS}_i &= \text{HMAC-SHA-1}(\text{MasterSessionKeyCS}, i) \\
\text{SessionID}_i &= \text{SHA-1}(\text{SessionKeySC}_i, \text{SessionKeyCS}_i) \\
\text{MasterSessionID} &= \text{SessionID}_0
\end{aligned}$$

Figure 3: *Sfsagent* and *rex* use the *MasterSessionKeys* and sequence number (*i*) to compute new *SessionKeys*.

to leave programs running that require use of the his *sfs-agent*. A utility *sfskey* lets the user list and manage open connections.

Once the master session has been established, the *rex* client can create subsequent secure connections (sessions) to the same server using the following protocol. First, *rex* contacts the *sfsagent* and requests a new session. The agent computes the values shown in Figure 3 based on the *MasterSessionKeys* (one for each direction) that were established using public-key cryptography during the initial connection. The *SessionKeys* are the symmetric keys that the *rex* client uses to encrypt its connection to *proxy*. They are computed as the HMAC-SHA-1 [7, 17] of a sequence number *i* keyed by the *MasterSessionKeys*. The agent generates a unique sequence number for each REX connection to prevent an adversary from replaying old REX sessions. The *SessionID* is a SHA-1 [7] hash of the *SessionKeys*, and the *MasterSessionID* is the *SessionID* where the sequence number is 0.

Once the *sfsagent* computes these values, it returns them to the *rex* client. *Rex* makes an insecure connection to *rex*d and sends the sequence number, the *MasterSessionID*, and the *SessionID*. Session IDs can safely be sent over an unencrypted connection because adversaries cannot derive session keys from them. *Rex*d looks up the appropriate cached connection based on the *MasterSessionID*. Then, *rex*d computes the *SessionKeys* and the *SessionID* for the new REX session (as in Figure 3) based on the sequence number that it just received and the *MasterSessionKeys* that it knows from the initial connection by the *sfsagent*. *Rex*d verifies that the newly computed *SessionID* matches the one received from the *rex* client. If they match, *rex*d passes the connection to *proxy* along with the new *SessionKeys*. Finally, *rex* and *proxy* both begin securing (encryption and message authentication code) the connection.

After *rex* and *proxy* establish a secure REX session, the *rex* client can create a new REX channel as described above. *Proxy* (and possibly also *rex*) will spawn the appropriate modules which can now communicate securely over the network. Subsequent connections proceed in the same way, allowing REX to rapidly execute processes on the server.

3 Extensibility

One of the main design goals for REX is extensibility. SSH has demonstrated that users want more features than just the ability to execute programs on a remote machine. TTY support, X11 forwarding, port forwarding, and agent forwarding, for example, are critical parts of today's remote execution tool. REX offers these features and also provides users with an interface to add new ones. REX's extensibility stems primarily from a single abstraction: the REX channel's ability to emulate file descriptor passing over the network. None of the features described in this section required any changes to the REX protocol.

3.1 TTY Support

REX provides pseudo-terminal support to interactive login sessions using the channel abstraction and file descriptor passing as follows. The *rex* client tells *proxy* to launch a module called *tttyd*, which takes as an argument the name of the actual program that the user wants to run. Typically, for remote login, the argument to *tttyd* is the user's shell.

Ttyd runs with only the privileges of the user who wants a TTY. The program has two tasks. First, it obtains a TTY from a separate daemon running on the server called *ptyd*. *Ptyd* runs with superuser privileges and is responsible only for allocating new TTYs and recording TTY usage in the system *utmp* file. The two processes, *tttyd* and *ptyd*, communicate via RPC. When *ptyd* receives a request for a TTY, it uses file descriptor passing plus an RPC reply to return the master and slave sides of the TTY. *Ttyd* connects to *ptyd* with *suidconnect*, SFS's authenticated IPC mechanism (described further in Section 3.4). This mechanism lets *ptyd* securely track and record which users own which TTYs.³ After receiving the TTY, *tttyd* keeps its connection open to *ptyd*. Thus, when *tttyd* exits, *ptyd* detects the event by an end-of-file. *Ptyd* then cleans up device own-

³Unlike traditional remote login daemons, *ptyd*, with its single system-wide daemon architecture, could easily defend against TTY-exhaustion attacks by malicious users. Currently, however, this feature is not implemented.

ership and `utmp` entries for any TTYs belonging to the terminated `ttyd`.

Once `ttyd` receives a newly allocated TTY, its second task is to spawn the program given as its argument (e.g., the user's shell). It spawns the process with the slave side of the TTY as its standard file descriptors and controlling terminal. Then, `ttyd` sends the file descriptor of the TTY's master side back to the `rex` client via the REX channel. On the client machine, `rex` copies data back and forth between this copy of the TTY's master file descriptor and the local terminal (e.g., the `xterm` in which `rex` was started).

`Rex` and `ttyd` also implement terminal device behavior that cannot be expressed through the Unix-domain socket abstraction. For example, typically when a user resizes an `xterm`, the application on the slave side of the pseudo-terminal receives a `SIGWINCH` signal and reads the new window size with the `ioctl` system call.

In REX, when a user resizes an `xterm` on the client machine, the program running on the remote machine needs to be notified. The `rex` client catches the `SIGWINCH` signal, reads the new terminal dimensions through an `ioctl`, and sends the new window size over the channel using file descriptor 0. Upon receiving the window resize message, `ttyd` updates the server side pseudo-terminal through an `ioctl`.

3.2 Forwarding X11 Connections

REX also supports X11 connection forwarding using channels and file descriptor passing. `Rex` tells `proxy` to run a module called `listen`, which finds an available X display on the server and listens for connections to that display on a Unix-domain socket in the directory `/tmp/.X11-unix`. `Listen` notifies the `rex` client of the display it is listening on by writing the display number to file descriptor 0.

Based on this remote display number, `rex` generates the appropriate `DISPLAY` environment variable that needs to be set in any X programs that are to be run. Next, `rex` generates a new (fake) `MIT-MAGIC-COOKIE-1` for X authentication. It sets that cookie on the server by having `proxy` run the `xauth` program. When an X client connects to the Unix-domain socket on the server, the `listen` program passes the accepted file descriptor over the channel to `rex`, which connects it to the local X server (i.e., it copies data between the received file descriptor and the local X server's file descriptor). `Rex` also substitutes the real cookie (belonging to the local X server) for the fake one.

3.3 Forwarding Arbitrary Connections

REX has a generic channel interface that allows users to connect two modules from the `rex` client command-line without adding any additional code. `Rex` creates a channel that connects the standard file descriptors of the server module program to a user-specified client module program. Unlike the channels described above, the `rex` client itself does not act as the client module. This generic mechanism allows REX users to easily build extensions such as TCP port forwarding and even SSH agent forwarding.

TCP port forwarding. Port forwarding essentially makes connections to a port on one machine appear to be connections to a different port on another machine. For example, a wireless network user concerned about eavesdropping might want to forward TCP port 8888 on his laptop securely to port 3128 of a remote machine running a web proxy. REX provides such functionality through three short utility programs: `listen`, `moduled` and `connect`. In this case, the appropriate `rex` client invocation is: `rex -m "listen 8888" "moduled connect localhost:3128" host`.

`Rex` spawns the `listen` program, which waits for connections to port 8888; upon receiving a connection, `listen` passes the accepted file descriptor over the channel. The `moduled` module on the server is a wrapper program that reads a file descriptor from its standard input and spawns `connect` with the received file descriptor as `connect`'s standard input and output. `Connect` connects to port 3128 on the remote machine and copies data between its standard input/output and the port. A web browser connecting to port 8888 on the client machine will effectively be connected to the web proxy listening on port 3128 of the server machine.

SSH agent forwarding. REX's file descriptor passing applies to Unix-domain sockets as well as TCP sockets. One useful example is forwarding an SSH agent during a remote login session. The `rex` client command syntax is similar to the port forwarding example, but reversed: `rex -m "moduled connect $SSH_AUTH_SOCK" "listen -u /tmp/ssh-agent-sock" host`.⁴ Here, the `-u` flag to the `listen` module tells it to wait for connections on a Unix-domain socket called `ssh-agent-sock`. Upon receiving a connection from one of the SSH programs (e.g., `ssh`, `scp`, or `ssh-add`) `listen` passes the connection's file descriptor to the client. The `moduled/connect` combination connects the passed file descriptor to the Unix-domain socket named by the environment variable `SSH_AUTH_SOCK`, which is where the real SSH agent is listening. In the remote

⁴When possible, `listen` rejects Unix-domain connections from other user IDs (through permission bits, `getpeereid`, or `SO_PEERCRECRED` `ioctls`). As this doesn't work for all operating systems, in practice we hide forwarded agent sockets in protected subdirectories of `/tmp/`.

login session on the server, the user also needs to set `SSH_AUTH_SOCK` to be `/tmp/ssh-agent-sock`. We have written a shell-script wrapper that hides these details of setting up SSH agent forwarding.

3.4 Forwarding the SFS agent

When first starting up, the *sfsagent* program connects to the local SFS daemon to register itself using authenticated IPC. SFS's mechanism for authenticated, intra-machine IPC makes use of a 120-line `setgid` program, *suidconnect*. *Suidconnect* connects to a protected, named Unix-domain socket, sends the user's credentials to the listening process, and then passes the connection back to the invoking program.⁵ Though *suidconnect* predates REX, REX's file descriptor passing was sufficient to implement SFS agent forwarding with no extra code on the server. Simply running *suidconnect* in a REX channel causes the necessary file descriptor to be passed back over the network to the agent on a different machine.

Once the *sfsagent* is available on the remote machine, the user can access it using RPC. All of the user's configuration is stored in one place; requests are always forwarded back to the agent, so the user does not see different behavior on different machines.

3.5 File system integration

One of the main motivations for building REX was to provide a remote execution tool that was integrated tightly with the SFS file system. When a user logs into a remote machine, he should see the same file systems as on the local machine. REX achieves this behavior by forwarding the *sfsagent*, which maintains a per-user view of the `/sfs` directory. Additionally, because the agent handles all of the configurable aspects of a user's environment—server key management, user authentication, revocation—the remote login session acts the same as the local one. SSH differs from this architecture in that an SSH user's environment might depend on the contents of his `.ssh` directory, which might be different between the local and remote machines.

4 Security

The REX architecture provides two main security benefits. First, REX minimizes the code that a remote attacker can exploit. Second, REX allows users to configure and manage trust policies during a remote login session.

⁵*getpeereid*, when available, is used to double-check *suidconnect*'s claimed credentials.

4.1 Minimizing exploitable code

In recent years, remote exploits have become a major concern for software developers. Buffer overruns and other bugs have led to serious system security compromises. REX attempts to mitigate this problem by minimizing the amount of remotely exploitable code. REX also attempts to protect against local exploits by minimizing the amount of code that runs with superuser privileges. REX offers protection against both types of exploits through the REX architecture's use of local file descriptor passing.

In REX, only *rex* listens for and accepts connections from remote clients. *Rex* runs with superuser privileges in order to authenticate the user (via *sfsauthd*) and then spawn *proxy* as that user. *Rex* uses local file descriptor passing to pass the client connection to *proxy*.

REX also tries to avoid local superuser exploits. For example, the privileged *ptyd* daemon allocates pseudo-terminals and passes them, using local file descriptor passing, to *ttyd* which runs with the privileges of a normal user. These privileged programs are small and perform only a single task, allowing easy auditing. Not counting general-purpose RPC and crypto libraries from SFS, *rex* is about 500 lines of code and *ptyd* is about 400 lines.

4.2 Managing trust policies

One particularly difficult issue with remote login is the problem of accurately reflecting users' trust in the various machines they log into. For example, a user may use local machine *A* to log into remote machine *B*, and then login from that session on *B* back to *A*. Many utilities support credential forwarding to allow password-free login from *B* back to *A*—but the user may not trust machine *B* as much as machine *A*. For this reason, other systems often disable credential forwarding by default, but the result of that is even worse. Users logging from *B* back into *A* will simply type their passwords. This is both less convenient and less secure, as an untrusted machine *B* will now not only be able to log into *A*, it will learn the user's password!

To address this dilemma, REX and the *sfsagent* support selective signing. Selective signing offers a convenient way to build up trust policies incrementally without sacrificing security. During remote login, REX remembers the machines to which it has forwarded the agent. In the remote login session, when the user invokes *rex* again and needs to authenticate to another server, his *sfsagent* will run a user-specified *confirmation program*. This program, which could be a simple text message or a graphical pop-up dialog box, displays the name of the machine originating the authentication request, the machine to which the user is trying to authenticate, the service be-

ing requested (e.g., REX or file system) and the key with which the agent is about to sign. The user's agent knows about all active REX sessions and forwarded agent connections, so the remote machine cannot lie about its own identity. Moreover, because signed authentication requests contain the name and public key of the server being accessed, as well as the particular service, the agent always knows exactly what it is authorizing.

With this information, the user can choose whether or not to sign the request. Thus, users can decide case-by-case whether to let their agents sign requests from a particular machine, depending on the degree to which they trust that machine. The modularity of the agent architecture allows users to plug-in arbitrary confirmation programs. Currently, SFS comes with a GUI program (see Figure 4) that displays the current authentication request and the key with which the agent is about to sign it. The user has five options: to reject the request; to accept (sign) it; to sign it and automatically sign all similar requests in the future; to sign it and all similar requests where the server being accessed is in the same DNS domain as the given server; and to sign it and all subsequent requests from the same client, regardless of the server being accessed.

5 Transparency

Due to the limited size of the IPv4 address space, machines often do not have static, globally routable network addresses. When an organization has more computers than IP addresses, it must typically resort to Network Address Translation, or NAT. With NAT, machines have private [25] (not globally routable) IP addresses on the local network, and a gateway re-writes the source address of any outgoing packets to be globally routable. The gateway then inverts this translation on any incoming packets, so it can deliver them to the right port on the appropriate local machine.

While NAT gateways let clients with private IP addresses connect normally to external machines, they have no analogous way of transparently supporting incoming connections to local servers. The reason is that most servers listen on well-known TCP or UDP ports. If the number of servers exceeds the number of globally routable IP addresses available, multiple server machines must share the same IP address, requiring some form of application-specific demultiplexing.

A related problem is that of dropped TCP connections. Sometimes the only globally-routable IP address available to a machine (or network of machines) is temporarily assigned and periodically changes. Also, laptops usually need to change IP addresses when transported between buildings. If one end of a TCP connection changes its IP address, the entire connection must be aborted.

NAT is another source of aborted TCP connections. Because NAT gateways must keep state for every active TCP connection, they can prematurely terminate a TCP connection when rebooted or when purging state entries for other reasons. Some NAT implementations (notably some cheap home routers optimized for web browsing) aggressively terminate TCP connections after only a few minutes of idle time.

Dropped TCP connections are particularly annoying with traditional remote login tools, as they cause the user's entire session to be aborted. Sessions may abort at inopportune times, when users are in the middle of editing files. Moreover all state associated with a dropped session is typically lost, including GUI windows forwarded from the remote machine.

Several design features allow REX to operate transparently through NATs and without fixed IP addresses. First, the SFS connection protocol allows servers to share IP addresses and even TCP ports, so that clients can connect transparently to arbitrarily many servers behind a NAT gateway with a single globally-routable IP address. Second, REX supports transparent resumption of aborted TCP connections [28, 39], so that a session need not be restarted after a change of IP address or NAT state flush.

5.1 Address and port sharing

The SFS framework, into which REX fits, provides two solutions for configuring servers behind NATs. The first approach, which we call address sharing, is to assign each internal SFS server a unique TCP port number. Most NAT gateways can be configured to have static mappings of external port numbers to private addresses and port numbers. For instance, TCP port 600 on the external IP address might always be translated to TCP port 4 of internal IP address *A*, while external port 601 is always mapped to port 4 on internal address *B*.

Though SFS servers by default listen on TCP port 4, a different port number can be specified with DNS SRV [12] records. Each SRV record maps an SFS server name and service to a server hostname (i.e., the name of the globally-routable IP address), a port number, and some priority information (so that multiple SRV records can be used for load balancing). In this way, the NAT administrator can configure an external TCP port for each internal SFS machine, and publish port numbers through DNS. External clients will then transparently connect to the appropriate port of the external address.

A second approach, which we call port sharing, requires only a single external TCP port number for all internal servers. All SFS protocols, including REX, begin with a CONNECT RPC in which the client specifies the desired self-certifying server name and service type (e.g., REX, file system, or authentication server). SFS's "meta-

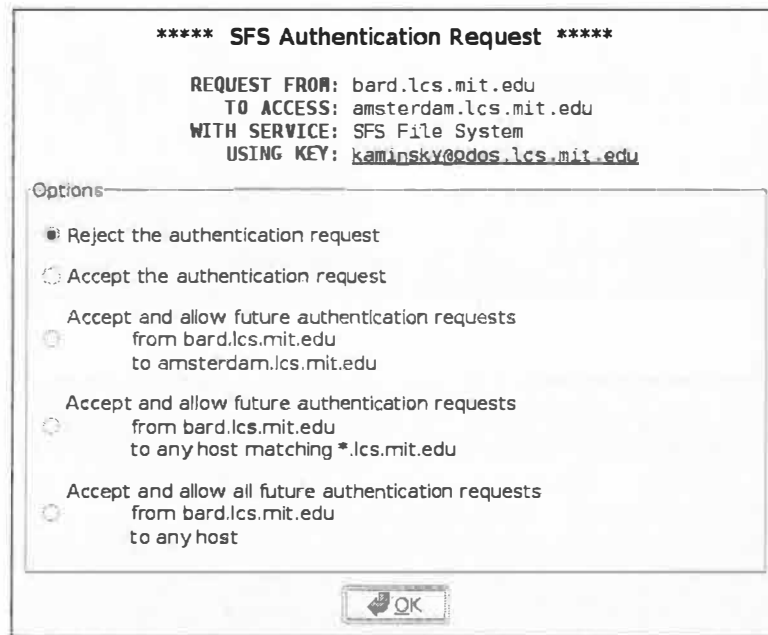


Figure 4: A GUI confirmation program

server” program, *sfsd*, can proxy TCP traffic to different internal IP addresses based on the contents of the initial CONNECT RPC. Port sharing with *sfsd* is similar to using the HTTP Host header with an HTTP proxy.

One advantage of port sharing is that *sfsd* can be configured to proxy certain services for a given internal server but not others (e.g., exporting an SFS file server but disallowing remote logins to it through REX). A security-conscious gateway administrator therefore has better control over what services are being made externally available. The disadvantage of port sharing is that its user-level TCP proxying consumes more CPU time and adds more latency than typical in-kernel NAT implementations.

A final issue with NATs is that, for efficiency reasons, machines on the internal network should connect to each other without going through the NAT gateway. The best way to achieve this goal is to run a split DNS server, which for the same hostname serves internal addresses to internal clients and external addresses to external clients. BIND and several other popular DNS servers support such functionality, but a number of users on the SFS mailing list have complained of the complexity of configuring DNS servers. Therefore, if split DNS is not available, DNS records can be set to point to the external IP address and internal machines can use a file `/etc/sfs/sfs_hosts` to override DNS with internal addresses. This file’s syntax is a superset of traditional `/etc/hosts`, extended to allow port numbers to be specified.

5.2 Session resumption

When a TCP connection aborts, REX provides the ability to resume the session over a new TCP connection. In order not to increase the amount of trusted or remotely exploitable code, this functionality is implemented entirely in *proxy*, with no changes required to *rex*. To resume an aborted TCP connection, the client first attaches to *proxy* through *rex*, using a new sequence number. It then issues a RESUME RPC, supplying the sequence number of the old session. This RPC causes the *proxy* to delete the state of the current session and replace it with that of the old session.

REX uses a bi-directional RPC protocol. Any input to *rex* prompts it to send an RPC to *proxy*, and similarly any program output to *proxy* results in an RPC to *rex*. For a resumable connection, *rex* and *proxy* each keep a replay cache of recently transmitted RPCs replies. Resumption then just consists of replaying all unanswered RPCs. In order to determine when something can be evicted from the replay cache, the RPC code conservatively determines when the other side has received a reply based on two factors: the size of the kernel’s TCP send buffer and replies to RPCs in the other direction.

One issue introduced by session resumption is the potential to leave stale proxies around if *rex* processes are terminated. REX employs several techniques to reduce the incidence of stale proxies. First, each *rex* client maintains a connection to the user’s agent. If a resumable *rex* process dies (for instance because the user terminates it with the Unix “kill -9” command), the agent detects

this fact by an end-of-file, and informs the remote proxy that the particular session can be garbage-collected.

Second, each agent has a unique identifier, based on the user's login name and the name of the machine it is running on. The agent's identity is supplied as a command line option to *proxy* (which, in particular, makes it visible through the Unix *ps* command). Whenever *proxy* is launched with a particular agent identity, it informs any previous *proxy* running with the same identity through a named Unix-domain socket in */tmp*, and the previous proxy then considers all sessions non-resumable. In the event that the agent ungracefully exits (for example, the client crashes and reboots), this mechanism causes the old proxy to exit the next time the user logs into the same server.

Session resumption works transparently even when the server changes IP address, so long as the server publishes its current address through DNS (e.g., using some sort of dynamic DNS scheme like *dyndns.org*). However, there are some subtleties to making this work properly because of the fact that DNS can also be used for load balancing—for instance, a hostname like *dialup.mit.edu* might actually point to a pool of login servers. In such cases, when a client changes IP address, it must resume its REX session to the same dialup server. To achieve this, REX revalidates all DNS information when reconnecting, and chooses the same DNS records as for the initial connection if still available. More precisely, when the original connection used an SRV record, if the particular hostname and port chosen the first time are still available, reconnection uses them again. For a given hostname, if the particular IP address initially used is still available, reconnection again re-uses it.

We note that the level of indirection provided by SRV records allows the location of an entire network of servers behind a NAT gateway to be updated with the change of a single DNS A (address) record. For example, Figure 5 shows an example of SRV records for four SFS servers in the static DNS domain *mydomain.org*, located behind a NAT gateway called *mynat.dyndns.org*. If the gateway's external address changes, only the DNS record of *mynat.dyndns.org* needs to be updated—the *mydomain.org* domain can remain unchanged.

6 Evaluation

First, this section quantifies REX's extensible architecture in terms of code size. Second, we compare the performance of REX with the OpenSSH [21] implementation of SSH protocol version 2 [37]. The measurements demonstrate that the extensibility gained from file descriptor passing comes at little or no cost.

6.1 Code size

REX has a simple and extensible design. Its wire protocol specification is only 230 lines of Sun XDR code [29]. REX has two component programs that run with enhanced privileges. *Rexd* receives incoming REX connections and adds only 500 lines of trusted code to the system (not counting the general-purpose RPC and crypto libraries from the SFS toolkit [19]). *Ptyd* allocates pseudo-terminals to users that have successfully authenticated and is about 400 lines of code.

Proxy runs with the privileges of the authenticated users and is just over 1000 lines of code; the *rex* client is about 2,350 lines. Extensions to the *sfsagent* for connection caching constitute less than 900 lines of code.

Modules that extend REX's functionality are also small. The *listen*, *moduled*, and *connect* modules are approximately 250, 30, and 375 lines of code, respectively. *Ttyd* is under 260 lines.

If REX were to gain a sizable user base, we could expect the code size to grow because of demands for features and interoperability. The code growth, however, would take place in untrusted components such as *proxy* or in new external modules (likely also untrusted). Because of the extensibility, well-defined interfaces, and the use of file descriptor passing, the trusted components can remain small and manageable.

6.2 Performance

We measured the performance of REX and OpenSSH 3.8p1 [21] on two machines running Debian with a Linux 2.4 kernel. The client machine consisted of a 2 GHz Pentium 4 with 512 MB of RAM. The server machine consisted of a 1.1 GHz AMD Athlon with 768 MB of RAM. A 100 Mbit, switched Ethernet with a 59 μ sec round-trip time connected the client and server. Each machine had a 100 Mbit Ethernet card.

We configured REX and SSH to use cryptographic systems of similar performance. For authentication and forward secrecy, SFS uses the Rabin-Williams cryptosystem [33] with 1,024-bit keys. SSH uses RSA with 1,024-bit keys for authentication and Diffie-Hellman with 768-bit ephemeral keys for forward secrecy. We configured SSH and SFS to use the ARC4 [14] cipher for confidentiality. For integrity, SFS uses a SHA-1-based message authentication code while SSH uses HMAC-SHA-1 [7, 17]. Our SSH server had the privilege separation feature [24] enabled.

6.2.1 Remote login

We compare the performance of establishing a remote login using REX and SSH. We expect both SSH and REX to perform similarly, except that REX should have

```

; SERVICE/NAME                                PRIO/WGHT  PORT  SERVER
_sfs._tcp.server-a.mydomain.org. SRV  0  1    600  mynat.dyndns.org.
_sfs._tcp.server-b.mydomain.org. SRV  0  1    601  mynat.dyndns.org.
_sfs._tcp.dialup.mydomain.org.   SRV  0  1    602  mynat.dyndns.org.
_sfs._tcp.dialup.mydomain.org.   SRV  0  1    603  mynat.dyndns.org.

```

Figure 5: An example of DNS SRV for four SFS servers on different TCP ports of `mynat.dyndns.org`. Such configurations are useful when `mynat.dyndns.org` is a NAT gateway, forwarding different TCP ports to different internal server machines. The *priority* and *weight* columns affect load balancing across duplicate records. The values are meaningless for `server-a` and `server-b`, and for `dialup` result in uniform distribution of connections across TCP ports 602 and 603 of `mynat.dyndns.org`.

a lower latency for subsequent logins because of connection caching.

Protocol	Average Latency	Minimum Latency
SSH	121 msec	120 msec
REX (initial login)	51 msec	50 msec
REX (subsequent logins)	21 msec	20 msec

Table 1: Latency of SSH and REX logins

Table 1 reports the average and minimum latencies of 100 remote logins in wall clock time. In each experiment, we log in, run `/bin/true`, and then immediately log out. The user’s home directory is on a local file system. For both REX and SSH, we disable agent forwarding, pseudo-tty allocation, and X forwarding.

The results demonstrate that an initial REX login is slightly faster than an SSH login. In both cases, much of the time is attributable to the computational cost of modular exponentiations. An initial REX connection requires two concurrent 1,024-bit Rabin decryptions—one by the client for forward secrecy, one by the server to authenticate itself—followed by a 1,024-bit Rabin signature on the client to authenticate the user. All three operations use the Chinese Remainder Theorem to speed up modular exponentiation.

An SSH login performs a 768-bit Diffie-Hellman key exchange—requiring two 768-bit modular exponentiations by each party—followed by a 1,024-bit RSA signature for server authentication and a 1,024-bit RSA signature for user authentication. The Diffie-Hellman exponentiations cannot be Chinese Remaindered, and thus are each more than 50% slower than a 1,024-bit Rabin decryption. The RSA operations cost the same as Rabin operations.

The cost of public key operations has no bearing on subsequent logins to the same REX server, as connection caching requires only symmetric cryptography. Were SSH to implement connection caching, we would expect performance similar to REX’s on subsequent logins.

6.2.2 Port forwarding throughput

Both SSH and REX can forward ports and X11 connections. To demonstrate that REX performs just as well as SSH, we measure the throughput of a forwarded TCP port with NetPipe [27]. NetPipe streams data using a variety of block sizes to find peak throughput.

Protocol	Throughput	Latency
TCP	87.1 Mbit/sec	59 μ sec
SSH	86.2 Mbit/sec	147 μ sec
REX	86.0 Mbit/sec	197 μ sec

Table 2: Throughput and latency of TCP port forwarding

We first measure the throughput of an ordinary, insecure TCP connection. Table 2 shows that the maximum TCP throughput is 87.1 Mbit/sec. The round-trip latency represents the time to send one byte of data from the client to the server, and receive acknowledgment. Next, we measure the throughput of a forwarded port over established SSH and REX connections. Table 2 shows that file descriptor passing in REX does not noticeably reduce throughput.

We attribute the additional latency of ports forwarded through REX to the fact that data must be propagated through both *proxy* and *connect* on the server, incurring an extra context switch in each direction. If *rex* and *proxy* provided a way to “fuse” two file descriptors, we could eliminate the inefficiency. Note, however, that over anything but a local area network, actual propagation time would dwarf the cost of these context switches.

7 Related Work

Several tools exist for secure remote login and execution. This section focuses primarily on those tools but concludes with a discussion of agents and file descriptor passing.

7.1 SSH

SSH [38] is the de-facto standard for secure remote execution and login. SSH is decentralized: one needs only local superuser privileges to run the SSH server daemon, and one does not need to obtain server certificates or otherwise register with any sort of realm administrator in order to connect to the SSH server. SSH also offers several modes of user authentication. For example, it has optional support for Kerberos [30], allowing password-free login plus ticket and AFS [13] token forwarding.

SSH was the main inspiration for REX, as we needed an SSH-like tool that could work with SFS. Though we could have extended SSH for the task, we decided to build REX from scratch for several reasons. First, we believed a design based on file descriptor passing would simplify implementation, improve security, and increase extensibility. Leveraging SFS's RPC compiler and library further reduced the amount of new code needed. We also wished to take advantage of SFS's infrastructure for user and server authentication, particularly its use of SRP to sidestep potential man-in-the-middle attacks. Finally, as commonly configured, SSH servers read files in users' home directories before authenticating them, which is inconvenient when the home directories themselves reside on SFS.

Aside from file descriptor passing and integration with SFS, REX offers several features not presently available in SSH. REX's connection caching improves connection latency. Connection resumption and support for NATs allow REX to operate transparently over a wider variety of network configurations. Selective signing improves security in mixed-trust environments and saves users from typing their passwords unnecessarily. Conversely, SSH provides features not present in REX, notably compatibility with other user-authentication standards.

We believe many of the ideas in REX are applicable to SSH and other remote login tools, and hope that SSH and REX can increasingly adopt each other's features. For example, as part of the privilege separation code in OpenSSH [21], the OpenSSH server internally handles pseudo-terminals with file descriptor passing. Though file descriptor passing is part of the source code, it is not part of the protocol. Generalizing the idea cleanly to pass file descriptors for other purposes would require modification to the SSH protocol, which we hope people will consider in future revisions.

7.2 Kerberos

Kerberos [30] is a centralized authentication system which includes remote login and execution utilities. It provides a unified way of naming, authenticating, and authorizing principals. Kerberos organizes users and machines into realms. Joining an existing realm (i.e., setting

up a server) requires permission from and coordination with that realm's trusted administrator. In part because Kerberos is based on shared-secret cryptography, creating a new realm is not a simple task and requires administrative permission to interoperate with existing realms.

Kerberized remote login is based on this centralized architecture, and therefore requires a trusted third party for client-server authentication. REX and SFS both support third-party authentication, but do not require it, and in practice they are often used without it. The AFS [13] file system uses Kerberos for authentication, and Kerberized remote login can authenticate users to the file system before logging them in. REX provides similar support for the SFS file system.

7.3 Globus

The Globus [8] Project provides a Grid metacomputing infrastructure that supports remote execution and job submission through a resource allocation manager called GRAM [5] and access to global storage resources through GASS [1]. Globus was designed to provide a uniform interface to distributed, remote resources, so individual client users do not need to know the specific mechanisms that local resource managers employ. By default, GRAM and GASS provide simple output redirection to a local terminal for programs running on a remote machine. Tools built on top of Globus can offer features such as pseudo-terminals, X11 forwarding and TCP port forwarding [11]. These features, however, seem to be built into the software and protocol whereas REX provides the same extensibility and security (privilege separation) through file descriptor passing.

The Grid Security Infrastructure (GSI) [3, 9] provides security and authentication to Grid-based services. GSI is based on X509 [36] public-key certificates and the SSL/TLS protocols [6]. Recent extensions to GSI add support for proxy certificates [32], which allow an entity to delegate an arbitrary subset of its privileges. A new GSI-enabled version of SSH can use these proxy certificates to provide limited delegation to applications running on the remote machine, similar to REX's selective signing mechanism.

7.4 Secure rlogin

Before SSH, researchers explored other options for secure remote login [16, 31]. Kim et al. [16] implemented a secure *rlogin* environment using a security layer beneath TCP. The system defended against vulnerabilities created by hostname-based authentication and source address spoofing. Secure *rlogin* used a modular approach to provide a flexible security policy. Like REX, secure *rlogin* used small, well-defined module interfaces.

REX uses a secure TCP-based RPC layer implemented by SFS; secure *rlogin* used a secure network layer between TCP and IP, similar to IPSec [15].

7.5 Agents

While REX is not the first remote execution tool to employ user agents, it makes far more extensive use of its agent than other systems. The SSH agent, for example, is capable of authenticating users to servers. For other tasks such as server authentication, however, SSH relies on configuration files (e.g., `known_hosts`) in users' home directories. When users have different home directories on different machines, they see inconsistent behavior for the same command, depending on where it is run. By contrast, encapsulating all state behind an RPC agent interface allows a user's configuration to be propagated from machine to machine simply by forwarding an RPC connection.

Another significant difference between the REX and SSH agents is that the SSH agent returns authentication requests that are not cryptographically bound to the identity of the server to which they are authorizing access. As a result, a remote SSH client could lie to the local agent about what server it is trying to log into. Concurrently and independently of REX, the SSH agent added support for a simple confirmation dialog feature, but the SSH agent is unable to build up any meaningful policies or even tell the user exactly what is being authorized.

Recently, the security architecture for the Plan 9 system has been redesigned [4]. The new Plan 9 architecture has an agent, *factotum*, which is similar to the SSH and SFS agents but is implemented as a file server.

The Taos operating system [18, 34] and the Echo file system [2] also have notions of an authentication agent. Unlike SFS, they both implement the agent as an operating-system component rather than as a user-controlled program.

7.6 File descriptor passing

An alternative to file descriptor passing is file namespace passing, as is done in Plan 9 [22]. Plan 9's CPU command can replicate parts of the file namespace of one machine on another. When combined with device file systems like `/dev/fd`, this mechanism effectively subsumes file descriptor passing. Moreover, because so much of Plan 9's functionality (including the windowing system) is implemented as a file system, CPU allows most types of remote resource to be accessed transparently. Unfortunately, Unix device and file system semantics are not amenable to such an approach, which is one of the reasons tools like SSH have developed so many

different, ad hoc mechanisms for handling different types of resources.

8 Conclusions

REX provides secure remote login and execution in the tradition of SSH. REX offers a new architecture with three main goals—extensibility, security, and transparent connection persistence in the absence of global routing. REX's extensibility, based on emulated file descriptor passing between machines, allows users to add new functions to REX without changing the protocol. REX's security benefits are a limited amount of exploitable code and a convenient mechanism for building trust policies. Finally, REX provides transparent operation in today's complex network configurations, which include NAT and dynamic IPs.

The current REX implementation demonstrates that the REX architecture is viable. We hope that the new ideas upon which REX is built will find wider applicability in other systems. REX is available as part of the SFS distribution (<http://www.fs.net/>).

9 Acknowledgments

We thank our shepherd Werner Vogels and the anonymous reviewers for their comments and feedback. Niels Provos provided helpful feedback on an early draft of the paper. This research was supported by the DARPA Composable High Assurance Trusted Systems program (BAA #01-24) under contract #N66001-01-1-8927, and by the National Science Foundation under Cooperative Agreement No. ANI-0225660 (as part of the IRIS project). Michael Kaminsky was partially supported by a National Science Foundation Graduate Research Fellowship, and David Mazières by an Alfred P. Sloan Research Fellowship.

References

- [1] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–88, Atlanta, GA, May 1999.
- [2] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Systems Research Center, Palo Alto, CA, September 1993.
- [3] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch. A national-scale authentication infrastructure. *IEEE Computer*, 33(12):60–66, 2000.
- [4] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan. Security in Plan 9. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

- [5] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.
- [6] T. Dierks and C. Allen. The TLS Protocol, Version 1.0. RFC 2246, Network Working Group, January 1999.
- [7] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.
- [8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.
- [9] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, San Francisco, CA, November 1998.
- [10] fsh — Fast remote command execution. <http://www.lysator.liu.se/fsh/>.
- [11] glogin. <http://www.gup.uni-linz.ac.at/glogin/>.
- [12] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, Network Working Group, February 2000.
- [13] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [14] Kalle Kaukonen and Rodney Thayer. A stream cipher encryption algorithm “arcfour”. Internet draft (draft-kaaukonen-cipher-arcfour-03.txt), Network Working Group, July 1999. Work in progress.
- [15] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, Network Working Group, November 1998.
- [16] Gene Kim, Hilarie Orman, and Sean O’Malley. Implementing a secure rlogin environment: A case study of using a secure network layer protocol. In *Proceedings of the 5th USENIX Security Symposium*, pages 65–74, Salt Lake City, UT, June 1995.
- [17] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Network Working Group, February 1997.
- [18] Butler Lampson, Martín Abadi, Michael Burrows, and Edward P. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.
- [19] David Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX*, pages 261–274. USENIX, June 2001.
- [20] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, 1999.
- [21] OpenSSH. <http://www.openssh.com/>.
- [22] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *ACM SIGOPS Operating System Review*, 27(2):72–76, Apr 1993.
- [23] Dave Presotto and Dennis Ritchie. Interprocess communication in the eighth edition UNIX system. In *Proceedings of the 1985 Summer USENIX Conference*, Portland, OR, 1985.
- [24] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.
- [25] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address allocation for private internets. RFC 1918, Network Working Group, February 1996.
- [26] Jerome Saltzer. Protection and control of information in multics. *Communications of the ACM*, 17(7):388–402, July 1974.
- [27] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performance evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.
- [28] Alex C. Snoeren. *A Session-Based Architecture for Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, December 2002.
- [29] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.
- [30] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*, pages 191–202, Dallas, TX, February 1988. USENIX.
- [31] David Vincenzetti, Stefano Taino, and Fabio Bolognesi. Stel: Secure telnet. In *Proceedings of the 5th USENIX Security Symposium*, pages 75–84, Salt Lake City, UT, June 1995.
- [32] V. Welch, I. Foster, C. Kesselman, O. Mulmo, S. Tuecke L. Pearlman, J. Gawor, S. Meder, and F. Siebenlist. X.509 proxy certificates for dynamic delegation. In *Proceedings of the 3rd Annual PKI R&D Workshop*, April 2004.
- [33] Hugh C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.
- [34] Edward P. Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.
- [35] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.
- [36] X.509. *Recommendation X.509: The Directory Authentication Framework*. ITU-T (formerly CCITT) Information technology Open Systems Interconnection, December 1988.
- [37] T. Ylönen and D. Moffat (Ed.). SSH Transport Layer Protocol. Internet draft (draft-ietf-secsh-transport-17.txt), Network Working Group, October 2003. Work in progress.
- [38] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.
- [39] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pages 95–106, Atlanta, GA, September 2002.

Network Subsystems Reloaded: A High-Performance, Defensible Network Subsystem

Anshumal Sinha Sandeep Sarat Jonathan S. Shapiro
anshumal@cs.jhu.edu sarat@cs.jhu.edu shap@cs.jhu.edu

*Systems Research Laboratory
Department of Computer Science
Johns Hopkins University*

Abstract

Traditionally, operating systems have used monolithic network stack implementations: implementations where the whole network stack executes in the kernel or (in microkernels) in a single, trusted, user level server. Code maintenance issues, ease of debugging, need for simultaneous existence of multiple protocols, and security benefit have argued for removing the networking implementation from kernel and dividing it into multiple user level protection domains. Previous attempts to do so have failed to deliver adequate performance. Given the advances made in both hardware (CPU, Memory, NIC) and micro-kernel design over the last decade, it is now appropriate to re-evaluate how these re-factored implementations perform, and to examine the reasons for earlier failures in greater detail.

Building on the primitives of the EROS microkernel, we have implemented two network subsystems: one a conventional, user mode, monolithic design and the other a domain-factored user level networking stack that restructures the network subsystem into several protection domains. We show that the restructuring maintains performance very close to that of the monolithic design, and that *both* designs compare favorably to a conventional in-kernel implementation. We discuss the issues faced in engineering the domain-factored implementation to achieve high performance, and present the quantitative evaluation of the resulting network subsystems.

1 Introduction

Traditionally, network systems have been implemented as monolithic subsystems that execute in the kernel or (in microkernels) in a single, trusted, user level server. To achieve performance, monolithic implementations sacrifice flexibility, maintainability, and security. Application specific knowledge cannot easily be incorporated into the network subsystem, debugging is more difficult, and the network stack itself becomes either a single point of failure for the entire system (in-kernel) or a single point of failure for the application (library approaches). Since network stacks are large, complicated software systems, they are intrinsically vulnerable to attack. It is therefore desirable to isolate *both* the operating system and the client application from the security vulnerabilities of the network subsystem.

Previous attempts to do so – most notably by Thekkath *et al.* [TNML93] – have generated disappointing results, suggesting that this design approach may be impractical. Unfortunately, Thekkath's analysis does not evaluate in detail the breakdown of time spent in various components. As a result, it is difficult to separate the impacts of three effects: user-level implementation, domain factoring, and the poor performance of the Mach Microkernel. Given the advances made in both hardware (CPU, Mem-

ory, NIC) and micro-kernel designs over the last decade, it is now appropriate to re-evaluate how such re-factored implementations perform, and to examine the reasons for earlier failures in greater detail.

Since in-kernel network stacks are the most common implementation approach, we use Linux [BC00] as a reference baseline for performance comparison. When a domain structured implementation is compared directly to an in-kernel implementation, the resulting comparison can be difficult to understand. In particular, it is difficult to know how to separate the performance consequences of user level implementation from the performance consequences of domain factoring. In order to support such evaluation, we have implemented *two* network stacks derived from a common code base: one monolithic, and the other factored into multiple protection domains.

Our monolithic implementation is a conventional microkernel network stack implemented as a user mode application. The network stack (based on lwIP [Dun02]) includes the ethernet drivers. Client application(s) and the network stack execute in separate protection domains. This implementation is roughly comparable to the conventional Linux implementation. Compromising the network stack compromises all network client applications, and also the entire kernel: most modern network interface

cards (NICs) implement physical DMA for performance reasons which implies that the network driver can overwrite arbitrary kernel memory.

The domain factored implementation places the ethernet driver in a separate protection domain, using a packet filter [MRA87] to demultiplex packets to the appropriate network stack. In this implementation, the network stack itself has no privileged access to the hardware. The impact of a compromised protocol stack is limited to a single application. The complexity of the packet demultiplexer (the ethernet driver) is primarily driven by the hardware interface, and can be validated independent of the protocol stack. The primary added cost of this implementation is the introduction of additional protection domain crossing delays.

Thekkath *et al.* [TNML93] measured a conceptually similar design, showing performance degradations of 39% and 20% for 10 Mbit and 100 Mbit ethernet implementations (see Table 1; we consider here only those results using packet sizes that conform to the ethernet specification). We show in the present work that this overhead can be reduced to 13% using an unoptimized kernel implementation. We believe that an optimized kernel would bring this performance to within 5%.

In this paper we present the design, implementation and performance of our restructured network subsystem on a modern, high-performance microkernel.

2 Objectives

Based on the above discussion regarding the limitations faced by an in-kernel network stack and the constraints needed to implement the domain factored design, we arrived at a list of goals for the domain factored network subsystem. In this section we discuss these goals and issues involved.

Ideally a network subsystem should meet several simultaneous goals:

- **Flexibility:** It should support the co-existence of multiple protocols that may be fine-tuned to exploit application-specific knowledge.
- **Resource Accountability:** Clients should be responsible for providing all the resources necessary to support their network activities. Buffers used to store network data must therefore be supplied by the client. This immunizes the stack from the potential denial of resource attacks.

- **Isolation:** QoS Crosstalk should be avoided to prevent clients from interfering with each other.
- **Resilience:** The network subsystem should be resilient to faults, both intentional or unintentional, which might have crept into the implementation of the network subsystem.
- **Performance:** In spite of isolating the network subsystem in its own protection domain, the network subsystem should deliver throughput and latency comparable to a conventional implementation.

Meeting all of these goals simultaneously is challenging. Monolithic network subsystems combine all resource management into a single protection domain, which compromises resource accountability and isolation. In-kernel protocol stacks are a single point of failure that may impact the entire kernel. User-mode monolithic stacks, as have been built for several microkernels, remain a single point of failure impacting set of applications that are using the network. While this single point of failure cannot be entirely eliminated (the packet filter is necessarily shared), its size can be substantially reduced. This allows quality assurance efforts to be focused more effectively.

When previous networking stacks have been split into multiple protection domains, performance has suffered. Protection domain boundaries usually imply data copies from one address space to another. Both the copies themselves and the cross-domain control transfer operations (IPCs) become a source of performance degradation. Because of these overheads, it is frequently asserted that protection carries intrinsic overhead.

Of the various user level network subsystems that have been created by researchers, Thekkath's work come closest to our design. Thekkath proposed a user level implementation using an in-kernel packet demultiplexer and transport protocols as user level libraries [TNML93]. The performance results from this work (Table 1) were disappointing. We believe that this is primarily due to faults of the Mach microkernel [GDFR90] that was used in Thekkath's experiments. The Mach microkernel interface was not flexible enough to provide full resource accountability, its cache performance was inadequate to support Thekkath's design [CB93], and its interprocess communication performance was significantly lower than current designs such as L4 [HHL⁺97] or EROS [SSF99].

Unfortunately, Thekkath's analysis does not evaluate in detail the time spent in various components. As a result, it is difficult to separate the impacts of three effects: user-level implementation, domain factoring, and the poor per-

System	Throughput(Mb/s)			
	User Packet Size(bytes)			
	512	1024	2048	4096
Ethernet				
Ultrix 4.2A	5.8	7.6	7.6	7.6
Mach 3.0/UX(mapped)	2.1	2.5	3.2	3.5
Thekkaths	4.3	4.6	4.8	5.0
DEC SRC AN1				
Ultrix 4.2A	4.8	10.2	11.9	11.9
Thekkaths	6.7	8.1	9.4	11.9

Table 1: Performance results reported by Thekkath *et al.*
Table reproduced with permission of the author.

formance of the Mach Microkernel. In order to provide a better understanding of these contributions, we have implemented two protocol stacks: one is a monolithic user mode implementation and the other is factored into multiple protection domains.

3 Monolithic Network Subsystem

Our monolithic network subsystem is a conventional microkernel network stack implemented at user level. The protocol suite is based on lwIP, which is a lightweight implementation of IP, UDP and TCP designed for low memory embedded systems. We chose lwIP as our initial protocol stack because it is simple and highly portable. Our ethernet drivers were created by adapting existing Linux ethernet drivers to operate at user level. Client applications execute in a protection domain separate from the network stack.

Our initial objective in creating the monolithic stack was ease of implementation. Since the stack is monolithic, simultaneous existence of multiple protocol stacks is impossible, and it is a single point of failure in the system. Resources cannot be accounted for as network buffers are allocated from a private stack pool instead of the clients being charged for them. Lack of resource accountability and multiplexing at a high level (Session layer) [Ten01] imply that there exists crosstalk between the clients using the stack. However, this implementation serves as a useful reference for comparison against other monolithic implementations. It also shares most of its code with the domain structured implementation (Section 4), allowing an “apples to apples” comparison between the two approaches.

The monolithic stack has two “helper” processes. The IRQ helper notifies the stack of newly arrived interrupts. A timeout helper is used to notify the stack that a timeout has occurred (e.g. TCP timeouts). The helpers are

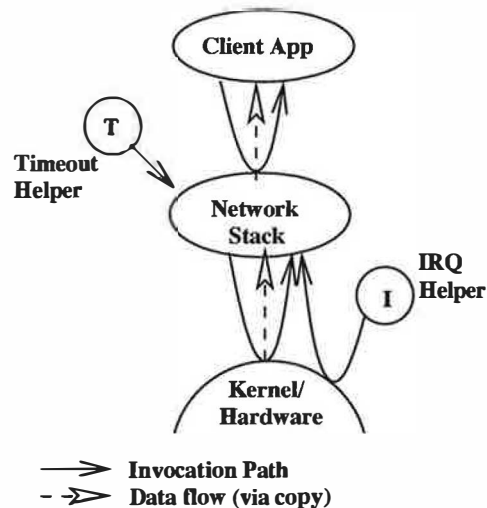


Figure 1: The EROS monolithic network subsystem

highly specialized and therefore small (<128 KB). The EROS kernel transparently allocates such small processes to small address spaces [Lie95] to reduce context switch overheads. As shown in Fig 1, the stack copies network data to/from the hardware DMA buffers to/from its buffer pool during network processing. Data flow across the client-stack process boundary is accomplished by a copy.

The performance of this implementation is comparable to the native Linux network stack, and is evaluated in Section 5.

4 Factored Network Subsystem

Like the monolithic implementation, the factored EROS network implementation is based on lwIP and executes in user mode. We present the design and implementation of the factored network subsystem and show how the designed goals are met.

4.1 Design

The factored version of the network subsystem has been structured to achieve all the design goals described earlier. We start by dividing the network subsystem into three independent protection domains namely, *enet* - consists of the ethernet drivers and a packet demultiplexer, *network stack* - consists of the various protocol implementations like IP, TCP, UDP, ICMP and ARP, and the network communication enabled *client application*. The division of the

network subsystem in this fashion is mostly at points of data multiplexing. We have effectively pushed the point of multiplexing down, adjacent to the network interface (into the enet layer). This is an accepted practice to minimize QoS crosstalk [Ten01].

Factoring makes it feasible to isolate attacks on the network subsystem and restrict the damage to the domain of attack, thus compartmentalizing the vulnerabilities of each domain. The result is a fault resilient, layered defense mechanism. Separating the enet layer from the network stack adds to the modularity of the design. This enables a client application to spawn a new stack in the event of a failure of the running stack. This also helps achieve flexibility as factoring supports simultaneous existence of multiple protocols.

4.1.1 Client-provided shared memory

Monolithic network subsystems use a centralized buffer pool. Network data meant for a client is buffered using buffers from this pool. So it is possible for a client to throttle the network bandwidth of another client using the stack without explicit resource management policies [DB96].

To accomplish proper resource accountability, it is necessary for the client to provide the store for the network data it requests/transmits. In the factored network subsystem, the client provides shared memory regions which are mapped three-way into the enet, the network stack and the client itself. This shared memory is used as a network buffer store by the stack for that particular client.

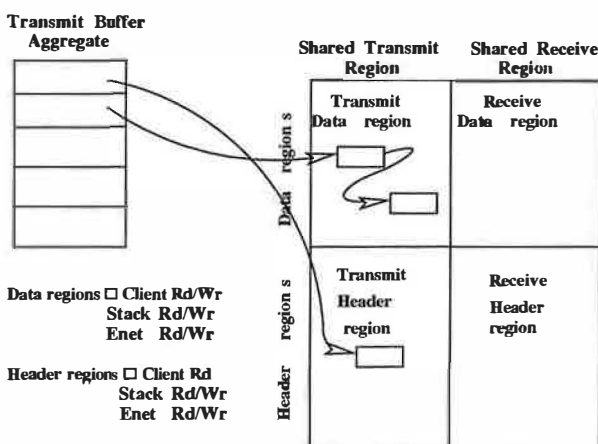


Figure 2: Client provided shared memory regions

The client is responsible for supplying four shared buffer

regions. They are:

- Transmit data region: Client specific transmit data is stored in buffers allocated from this region.
- Transmit header region: Client specific transmit protocol header is stored in buffers allocated from this region.
- Receive data region: Client specific received data is stored in buffers allocated from this region.
- Receive header region: Client specific received protocol header is stored in buffers allocated from this region.

The client has (read,write) permissions to only two of these regions - the transmit data region and the receive data region. The other two regions - the receive header region and the transmit header region are read only for the client. All regions are mapped into the enet and the network stack, with (read, write) permissions. (Figure 2).

The data regions are used as a store to allocate network buffers which contain client-specific data. The header regions are used as a store to allocate network buffers which contain protocol headers. A malicious client cannot manipulate or corrupt protocol headers as it has only read permissions to the buffers containing the header. Hence, it cannot influence the protocol state machine of the stack.

4.1.2 Scatter-Gather

Additional process boundaries add an extra cost to the data movement across these boundaries. We use shared memory to avoid copying across protection domains. Scatter-gather is used to enable copy avoidance. We describe this mechanism in detail for inbound and outbound network packets.

For outbound network traffic, the client acquires an unused network buffer from the transmit data region and places the data to be transmitted in it. The stack uses a free buffer in transmit header region to write the protocol headers and then chains these two buffers together into a single buffer chain. The enet uses the transmit descriptor to this buffer chain to transmit data.

The case of inbound network traffic is exactly inverse. The enet "scatters" the network packet into client specific data, storing it into buffers from the receive data region, and into protocol headers storing it into buffers from receive header region.

Since the buffers used for storing data come from the client, eager demultiplexing [DB96] is necessary at the enet level. As mentioned earlier, enet has a packet filter, which is used to identify the client to which the incoming data belongs. If the enet is unable to allocate network buffers from either of the two receive regions due to exhaustion, we simply drop the packet as is done in lazy receiver processing. The justification behind this policy is that ethernet does not provide guaranteed packet delivery in any case. If the client cannot keep up with the pace of incoming data, the ethernet driver is free to discard packets.

Resource accountability using the shared memory design presented can avert a potential denial of resource attack in which a rogue application can request network data and then refuse to dequeue its packets. Refusal to do so deprives other needy applications of precious network buffers. This attack is no longer feasible in the factored network subsystem as the buffers for the packets are allocated from the client's store. The client's refusal to dequeue the packets will only lead to client's buffer pool exhaustion. Once exhausted, the packets meant for that particular client are dropped by the enet. No other client can be affected due to the mis-behavior of the rogue client.

One advantage of this resource accounted shared memory design besides security is the *scatter-gather* mechanism which ensures a zero-copy, and hence helps in performance enhancement. Banga *et al.* [BDM99] only account for the execution time spent in the network stack on behalf of the client. With the factored network subsystem design, it is possible to extend the notion of network resource to include the buffers used for packet transfer also. This ensures that a client application is not able to deprive other clients of network buffer resources as in a centralized buffer pool design. This, in combination with the layered structure of the factored network subsystem, prevents the QoS crosstalk as the multiplexing is pushed further down in the network subsystem and the effect of one client on another is minimized.

4.2 Implementation

We have modified lwIP keeping in mind the design goals discussed earlier in Section 2. lwIP uses a stack-centralized pool of network buffers. We modified lwIP to use shared memory buffers. Enet includes a packet filter (adapted from the LRP implementation [DB96]) which is used to eagerly demultiplex packets to the appropriate receiving network stack.

We now describe the components which are handled dif-

ferently when compared to existing user level network implementations.

4.2.1 Shared Memory

The increased number of process boundary crossings in a factored design results in an increase in latency incurred during the processing. The main source of this latency is the cost of cross-space control transfers and cross-space data copies. To avoid these expensive data copies, shared memory is employed. In most existing implementations, the shared memory for the network buffers is a global entity allocated by some component of the network stack or is a memory pinned resource. In our factored design, the client provides a source of storage that is used to allocate a shared memory region that is used exclusively on behalf of that client. We list the steps involved in the creation of a shared buffer:

1. The client grants a storage allocator to the stack. The client can rescind this storage, but has no access to pages that are allocated using this allocator.
2. The stack allocates the four shared buffer regions that we described earlier using the storage allocator.
3. The buffers in the transmit regions are reserved for transmission exclusively and the buffers in the receive region for reception exclusively. The stack 'formats' these pages as ring buffers (described in the next section).
4. The stack requests the enet to map read/write version of all these pages into its address space.
5. The stack requests the client to map read/write version of the pages of the transmit, receive data region into its address space. Note that the client has only read access to the buffers in the header regions.

The mapping of a client-specific shared memory region into the various domains is now complete. This mapping is done during setup time when the client registers with the stack and the enet domains. The header region buffers (Figure 2) are used for buffering protocol headers and the data region buffers for client specific data. In the case of transmission, the client places data into the transmit data region buffers. The stack prepares protocol headers in transmit header region buffers and links the respective buffers together. This buffer chain is passed off to enet for transmission. The scenario of reception is exactly the inverse. A rogue client can at most mangle the buffers in the data region but has no access to the buffers in the header region. Hence, in no way can affect the stack or the enet domains.

4.2.2 Ring Buffers

We now describe the data structure used to hold the network buffers themselves. For ease of presentation, we only discuss the scenario during reception. The case of transmission is exactly the inverse.

As already described, buffers for client data are allocated from the data regions, while the protocol header is allocated from header regions. We initially format (Step 2 in the creation of the shared memory region) the four shared buffer regions into uniformly sized ring buffers. Each buffer consists of a buffer header (not to be confused with the packet header) and space which stores the buffer payload (network headers and network data reside here). The buffer header stores meta-data, including the status of the particular buffer, the size of payload stored in the buffer and the pointer to the next buffer (See figure 3).

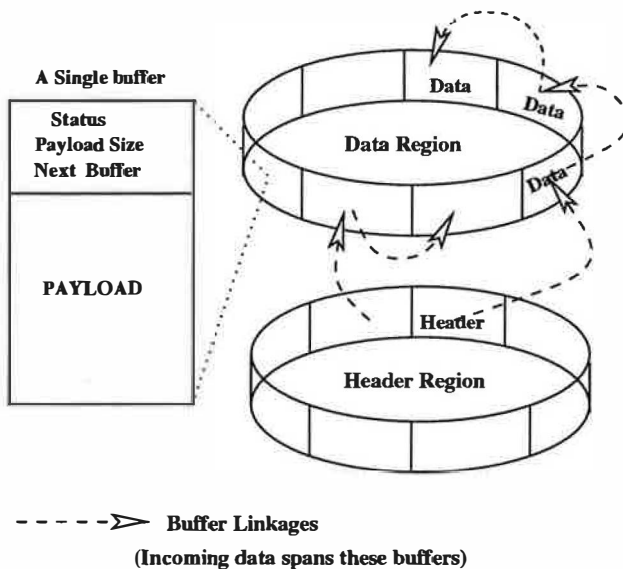


Figure 3: Receive ring buffers. The incoming data packet is scattered into header and client data and stored into appropriate ring buffers by the enet packet filter.

When a packet arrives for a particular client, the enet packet filter splits the packet into buffers allocated from two different ring buffers. Buffers for the packet data are allocated from the receive data region ring buffer. Buffers for the protocol headers are allocated from the receive header region ring buffers. Traditionally, the functionality of the packet filters in the ethernet layer is to demultiplex the incoming packet to the recipient stack [EK92]. We

have extended the packet filter to include placement of the inbound packet into the appropriate client and stack data and header ring buffers. Packets can appear on the network interface out of order and it is up to the higher layer protocols to re-order them. Buffer pointers can be appropriately redirected to make the data appear “contiguous”.

The ring buffer mechanism employed is similar to the hardware DMA ring buffer mechanism used by most network interfaces. A direct consequence of using ring buffers for the network packet is that allocation and deallocation occur in strictly increasing ring sequence and hence their time complexity is $O(1)$. A network buffer list implementation also achieves $O(1)$ allocation and deallocation. But it is slower, as it needs explicit mutexes to read/write the buffers.

4.2.3 Inter Domain Communication

We have extensively used the EROS IPC mechanisms, specifically *call*, *return* and *retry*. Equivalents to CALL and RETURN are commonly available in most micro kernels. The RETRY operation is specific to EROS. Clients queued using RETRY do not block other requests, and their reactivation honors the scheduling policy of the operating system.

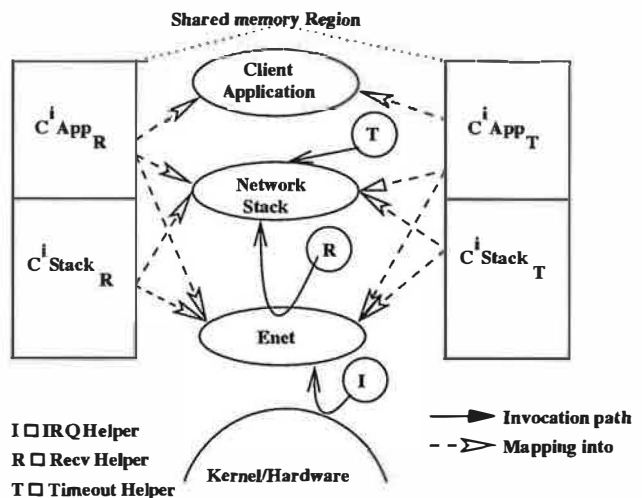


Figure 4: The three domains and Helpers. The client specific shared memory regions are shown too

EROS uses a synchronous IPC mechanism. Since calls are blocking, upcalls are prohibited as they can lead to denial of service attacks. If the enet makes an upcall to

the stack and the stack fails to return, the enet remains waiting indefinitely. Further upcalls can also lead to a potential deadlock where the stack and the enet are attempting to call each other simultaneously. So we introduce a few domains (Helpers) which have highly specific roles (Figure 4). They are:

- **IRQ helper:** Notifies the enet domain of newly arrived interrupts from the network interface. This process runs in an infinite loop waiting for an IRQ and on an IRQ signal turns around calls the enet layer.
- **Rx helper:** Notifies the stack that the enet has received data. This process sits in an infinite loop calling the stack and the enet in turns. The enet issues a RETRY to block Rx until data is available. The Rx helper relieves the network stack of having to block for I/O data so that client applications can avail the stack's services promptly.
- **Timeout Agent:** Notifies the stack of timeouts (e.g TCP timeouts). This process runs in an infinite loop calling the stack after regular time intervals (say 100 msec).

Separating the protocol processing, packet demultiplexing and applications into different threads has a performance disadvantage. By doing so, we have effectively replaced what would have been function calls in a monolithic design with more expensive IPC calls. Adding the helpers could potentially degrade our performance further.

Decreasing the latency in processing can be pursued by reducing the number of IPC invocations and the context switch time. We carry out a number of optimizations to this effect.

- The Helpers are specialized and small (<128KB). The EROS kernel transparently allocates such small process to small address spaces. The advantage is that unlike typical address space switches we can avoid TLB flush while switching to/from a small address space.
- “Amortizing” the IPC invocations that are needed so that we can reduce the per packet address space switches. We describe this in the next subsection.

4.2.4 Ping-Pong design

In the receiving data path we typically incur the following address space switches triggered by IPC : IRQ helper

→ enet (CALL enet IRQ notification), enet → IRQ helper, enet → Rx Helper (RETRY Rx Helper to notify the stack), Rx Helper → stack (CALL stack packet reception Notification), stack → Rx Helper (RETURN), Rx Helper → enet (CALL) (Figure 5). This is quite a high overload for a single packet. High speed NICs can generate thousands of interrupts per second. The CPU cannot keep up with this rate of interrupt generation. This can lead to *live-lock*. In this state, the system spends all of its resources processing incoming network packets, only to discard them later because no CPU time is left to service the receiving application programs.

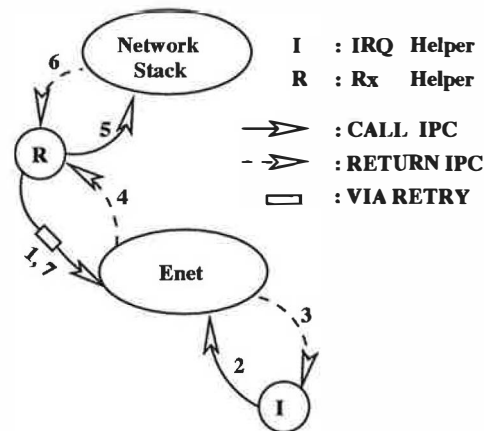


Figure 5: The various IPC invocations in the Rx data path. The numbers show the sequence of events.

Linux uses the NAPI poll approach [SO01] where interrupts are disabled and polling is scheduled for a certain time interval before re-enabling the interrupts. We use a similar flow-aware solution to tackle this. The rationale is that if the enet, stack, Rx Helper are already doing their respective tasks no IPC invocations are necessary for notification. Accordingly, the enet and stack processes ping-pong between receiving and transmitting. The enet alternates between reading a network packet from the hardware Rx ring buffer and writing a packet into the hardware Tx ring buffer. The stack alternates between reading data off the shared memory Rx ring buffers and writing data into the shared memory Tx ring buffers. The state of maximum throughput is achieved when both the stack and the enet are ping-ponging simultaneously between transmitting and receiving. In this state no IPC invocations are issued and the stack processes whatever the enet receives and the enet transmits the data on the transmit ring buffer named by the transmit descriptors. The only overhead in the data path is the context switch time between the stack and the enet (the IPC overhead is absent). The network interrupts are disabled until the enet stops ping-ponging.

5 Evaluation

This section presents both a quantitative evaluation of the various implementations and a qualitative evaluation of the resilience achieved in the factored implementation.

5.1 Performance

Performance measurements were carried out using two 900 MHz Pentium IIIs with 256 MB RAM and 33 MHz PCI bus, each equipped with a 3com 3C905C-TX (100Mbps) ethernet card and a NetGear 602T - BCM5701 (1000Mbps) gigabit card, interconnected by standard Cat-5e patch cables. The linux drivers for these NICs were ported to EROS. To fit our design we removed the NAPI poll interrupt mitigation approach to mitigate interrupts that linux uses in its gigabit driver.

Due to unrelated research that is ongoing in our laboratory, the version of EROS reported here does not provide an optimized IPC implementation. On the kernel used, a typical EROS round trip IPC takes 2268 cycles (2.40 μ s on our test machine). For comparison, the optimized implementation takes approximately 500 cycles per round trip (0.53 μ s). Transfers using smaller packet sizes are significantly influenced by IPC performance in all implementations, and we expect they would improve when using an optimized kernel.

We used the standard network tests ping and `ttcp` for benchmarking, considering the linux 2.4 network stack as a touchstone. We rate the performance of the refactored network subsystem and a monolithic version of the network subsystem (running lwIP) on EROS. lwIP running on linux is not listed here because it is severely crippled, as it uses a tuntap interface over linux.

5.1.1 Ping

The ping time between two hosts is a good measure of the latency. We measured the round trip latency time of ICMP echo requests with size varying from 64 bytes to 9000 bytes (fig 6).

Linux and the EROS-monolithic stack have comparable ping latencies throughout. EROS-monolithic is slightly slower because it uses the services of an IRQ notifier between the monolithic subsystem and the kernel. This is primarily an IPC-related delay. The EROS-factored implementation has a higher latency due to the increased number of address space switches as a result of the higher

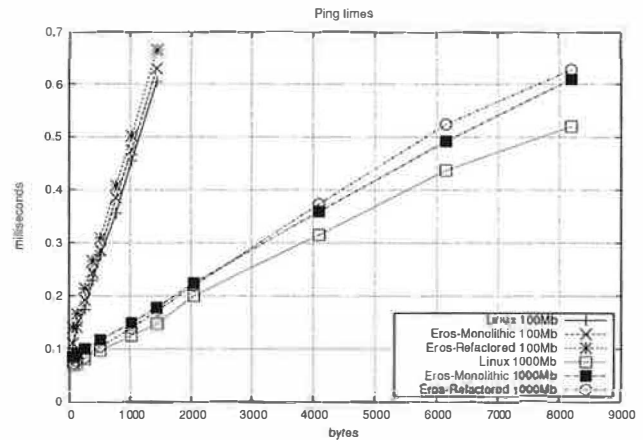


Figure 6: Ping times for the various stacks

number of processes (primarily the receive helper).

5.1.2 ttcp

Ttcp is a benchmark that measures the time taken to transmit and receive data between two systems using the UDP or TCP protocols. We ran `ttcp-r` for different sizes of socket buffers. We set up a `ttcp` transmitter on a redhat 2.4 linux machine and the receiver was on the candidate to be benchmarked. The maximum wire capacity on a 100 Mbps ethernet is 12.5 MBps and on a 1000Mbps Gigabit is 125MBps.

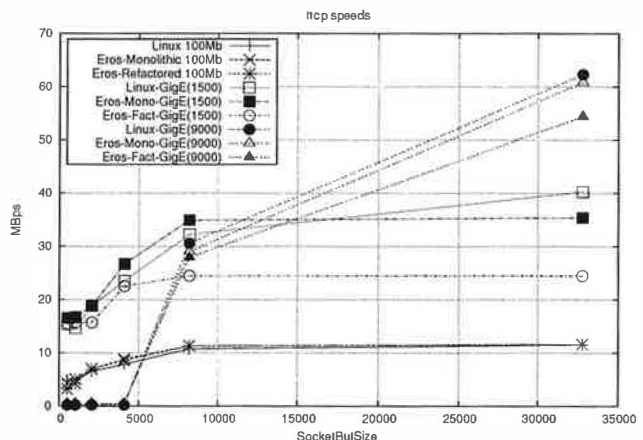


Figure 7: Observed ttcp speeds

Linux and the EROS-monolithic stack perform comparably in this test too. The Linux gigabit stack is slightly faster because of the NAPI interrupt mitigation in their driver. The EROS gigabit driver does not use this technique. Also the EROS-monolithic uses the services of an IRQ notifier between the monolithic subsystem and the kernel. As the socket buffer size decreases linux pays an increasing performance penalty due to the higher number of data copies from kernel to user space and back required for the same size of data to be transferred. EROS-monolithic incurs a similar cost copying the data from the user-mode network stack to the application using IPC.

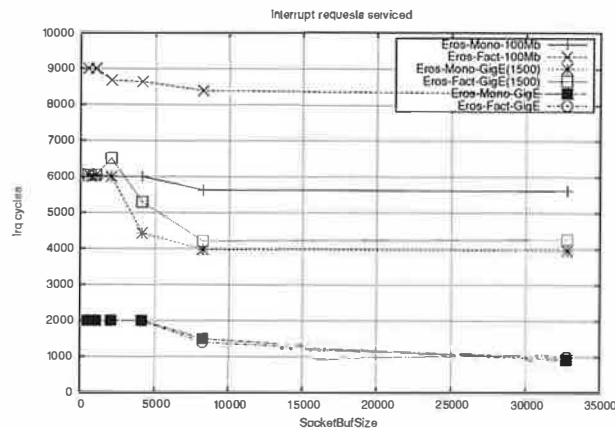


Figure 8: IRQs per 10,000 packets serviced during tcp.

Figure 8 and Figure 9 show the IRQs serviced and the user cycles consumed respectively during a typical tcp run. As seen from Figure 8, the factored network subsystem takes more number of interrupts than the monolithic version. This is due to the fact that in the monolithic stack, the protocol processing occurs in the context of the interrupt service routine (ISR). During protocol processing, packets that arrive do not flag interrupts and are serviced in the same context subsequently. In the factored network subsystem, the protocol processing and ISR occur in different processes viz. the stack and the enet. On packet arrival, the enet signals the stack to carry out packet processing and itself becomes available to receive new interrupts. The higher interrupt rate of the factored subsystem results in higher processor usage as seen from the user cycles figure (Figure 9).

The factored network subsystem achieves 87% performance of the monolithic linux stack (worst case) and up to 89% performance of the monolithic EROS stack. While the EROS-factored implementation avoids data copy overhead using shared memory, it incurs a larger number

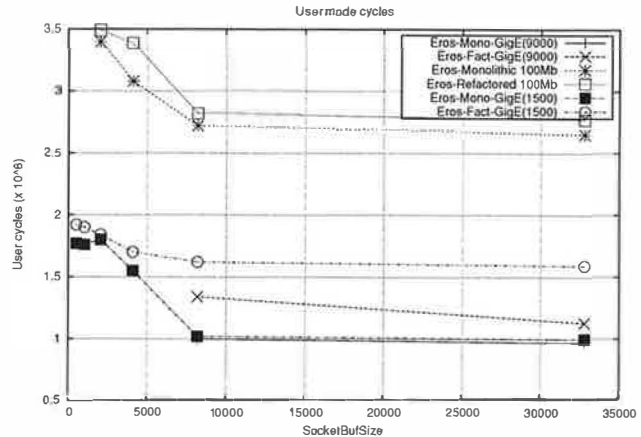


Figure 9: User cycles per 10,000 packets consumed during tcp. Data for GigE(9000) for socketbuf lengths less than 8192 have been omitted to avoid graph compression. Measurements for smaller socket buffer sizes for the Gigabit (9000 mtu) case are limited by Linux transmit performance.

of context switches for control transfer purposes. The RETRY operation in particular is excessively expensive.

Chen *et al.* [CB93] have argued that cache performance is an insurmountable obstacle to microkernel performance in general. While this claim was convincingly rebutted by Liedtke [Lie96], Blackwell has pointed out that instruction cache locality plays a significant role in small packet processing [Bla96], and Druschel's work on Fbufs [DP93] was motivated in part by cache-based domain crossing costs in the *x*-kernel. Mossberger *et al.* identify a number of important cache-related optimizations for reducing network processing latency [MPBO96].

Figure 10 compares instruction and data cache misses in the monolithic and factored EROS implementations. The results suggest that there is a relationship between instruction cache traffic and performance, but it is not a simple relationship. In some cases, the instruction cache costs are noticeably *higher* on the monolithic network stack, even though the monolithic implementation has slightly better performance in those cases. Note also that instruction cache effects dominate data cache effects overall by three decimal orders of magnitude in conventional ethernet frames. This is a pleasing result, both because instruction cache tuning is relatively easier to accomplish and because we know that the lwIP implementation objective was simplicity rather than performance. More detailed in-

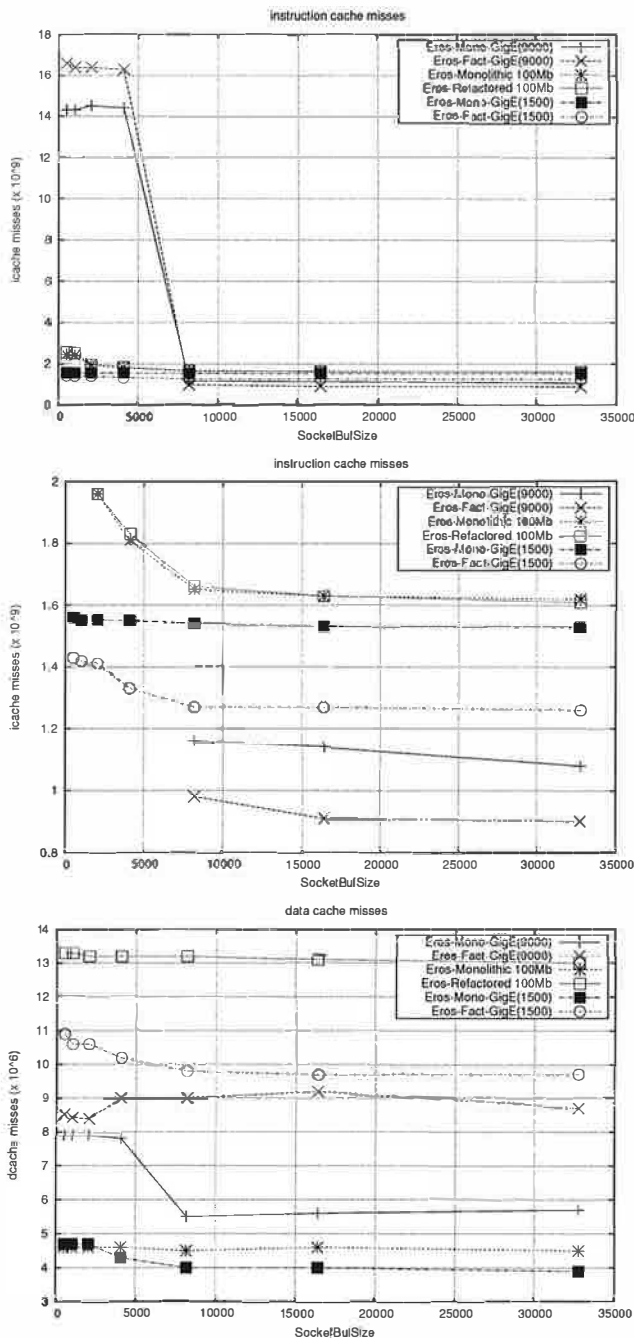


Figure 10: User instruction and data cache misses per 10,000 packets during tcp. The second graph is identical to the first with selected points removed to eliminate compressed display.

vestigation of these results is indicated.

We believe that higher performance would be achieved using the optimized IPC. Further, we note that lwIP was developed for low memory usage rather than for high performance, which suggests that further performance enhancements are possible in the protocol stack itself. However, the key lesson in these figures is that the performance throughout is directly proportional to the interrupt service rate. Our current implementation does not poll the network at high packet rate. Performance is therefore limited by the interrupt latency.

Microbenchmarks provide reliable measurement of low-level performance, but often fail to accurately predict the behavior of the overall system. End to end performance is heavily influenced by issues of locality, and domain factoring raises the risk of defeating these performance gains. Given this, the microbenchmark results reported here must be interpreted with caution, and may not prove to be definitive.

5.2 Resilience

A monolithic network stack suffers from several shortcomings that impact resilience:

1. Because it is monolithic, the network stack is a single point of failure for all applications using the network. In an in-kernel implementation, bugs in the network stack manifest as failures of the system as a whole.
2. Achieving precise per-client resource accountability is exceedingly difficult.
3. The amount of robustness-critical code (in lines) is large, which increases the intrinsic vulnerability of the network stack.
4. There are no well-defined internal interface boundaries at which fail-fast and recover-fast disciplines can naturally be applied.

Improvements on the first three criteria can be measured directly using straightforward metrics. The last requires performance evaluation.

The factored network stack presented in Section 4 is both a user-mode stack and a per-client stack. Being user-mode, it is unable to crash the system as a whole. Being per-client, failures in one stack do not impact the behavior of other client applications.

Several previous network stacks have provided CPU resource accountability and used eager demultiplexing for purposes of maintaining quality of service [Ten01, DB96]. The per-client network stack described here runs from a client-supplied scheduling class, ensuring that CPU allocation for network processing purposes observes the same scheduling policy as all other processing that occurs on behalf of an application. Novel to the factored design presented here is that storage resources are *also* allocated using per-client resources: each client provides a storage source from which its buffers are allocated.

The monolithic EROS stack described in Section 3 includes a total of 10,803 critical lines of code. Of these, 5,217 are the ethernet driver, and 5586 are the lwIP implementation. Any failure at any point in this code potentially compromises all networking applications. In contrast, the factored implementation of Section 4 places the lwIP implementation outside of the trusted code base, reducing the amount of critical code to 5,612 lines (the ethernet driver), leaving an untrusted lwIP implementation of 6,297 lines. All of these lines are in addition to the EROS kernel itself, which is approximately 15,985 lines of code. In contrast, the Mach kernel used in Thekkath's measurements was over 100,000 lines of code *excluding the network stack and drivers*. While precise sizes for Thekkath's stack are not available, the total critical component size of the stack described here are less than 16% of the earlier effort.

The fault recovery strategy for the factored network stack is first to fail eagerly. Failure may be signalled by various sorts of anomaly and intrusion detectors; choosing a specific mechanism for detection is beyond the scope of this paper. The interested reader may wish to examine [CPM⁺98] and its citations. Once a failure has been recognized, the relevant network subsystem is involuntarily halted and a new network subsystem is instantiated to replace it. Running on a 931Mhz Pentium-III, instantiation of a new network stack is accomplished in 387ms. This number is severely impacted by the currently unoptimized IPC implementation. Based on earlier measurements, a factor of two improvement would be expected in a production-suitable kernel.

5.3 Lessons for Microkernel Design

One lesson to draw from the factored network stack is the need for non-blocking notification mechanisms in microkernel architectures. The RETRY system call was originally introduced as a way to work around the absence of such a mechanism in the EROS design. It provides a means to enqueue a client in such a way that the client will

re-execute its last CALL when unblocked. This, coupled with the use of worker threads, can be used to simulate non-blocking notification. The RETRY operation design attempted to satisfy a philosophy, shared with Liedtke [Lie93] and Ford [FL94], that IPC operations should be thread migrating, blocking, and usually non-preemptive.

However, the end result is cumbersome to use, difficult for the programmer to understand, and introduces unnecessary context switches. Future derivatives of the EROS kernel will incorporate a non-blocking notification mechanism whose behavior is similar to that of hardware interrupts. In contrast to UNIX signals, delivery of notifications will be deferred until the recipient process next enters a "ready to receive" state, and their semantics and prioritization will be entirely determined by the recipient.

6 Related Work

A great deal of research has gone into building user level network subsystems. Several of them have relied on specialized hardware support in which network buffer pools on the card can be reserved by applications. HP developed such a mechanism for the Jetstream LAN [EWL⁺94] where applications could reserve buffer pools on the AfterBurner [DWB⁺93] card. The data on arrival was demultiplexed to the correct application pool. Unet [vEBBV95] implementation for SBA 200 modified the firmware to add a new Unet compatible interface.

The factoring of the subsystem into layers is an extension of the design proposed by Thekkath, U-net and Mogul [MRA87] who have all argued for separating the interface driver from the protocol stack. They have also extensively used shared memory between these layers to mitigate copy costs.

U-net follows a similar approach for a parallel and distributed computing architecture. Although Unet showed promising results performance-wise, it failed to address certain vulnerability issues. The shared memory buffers allocated to buffer network data were typically pinned to physical memory. This made them a scarce resource, thus making it vulnerable to possible denial of resource attacks. Using shared memory in network subsystem has been widely used concept. Druschel's work on Fbufs [DP93] uses the notion of transferrable buffers to reduce domain crossing overheads in the *x*-kernel. IOLite [PDZ99] uses shared memory and ACLs to ensure a protected unified buffering scheme. Exokernel [EKO95], which supports application level resource management scheme also refers to the design of sharing the network

buffers similar to the one suggested by Druschel [PDP94]. But none of the above mentioned works talk about the issues of making the applications accountable for the buffer they use. The EROS factored implementation requires that clients supply the resource for all buffers used on their behalf. It is this concept of resource accountability that makes EROS-factored design different from the above mentioned works.

We use eager demultiplexing and lazy processing in our design. Demultiplexing immediately at the network interface is necessary for purposes of QoS [Ten01, DB96] and for user-level implementation of network subsystems. This also accomplishes proper time resource accountability. In our scheme, space resource accounting is also accomplished. If the client is devoid of space for the network data, we simply drop the packet i.e. the stack does not waste time in protocol processing. The reason for this policy is that if the client is not able to keep up with the pace of incoming data, there is no need for the stack and driver to do that on behalf of the client.

We introduce a novel invocation pattern between the various domains in the network to improve the speed of the stack, by amortizing the cost in the invocation and interrupt servicing. Previous mechanisms [KMY01, SO01] use interrupt mitigation or interrupt coalescing. At high speeds, the processor cannot keep up with 1 interrupt per packet. The interrupt mitigation schemes work by disabling interrupts when there is work and re-enabling them when there is none. These schemes are application inconsiderate and need changes to the driver. Soft timers [AD00, ST93] allow efficient scheduling of software events. These can avoid interrupts and reduce context switches associated with network processing. But these are operating system facilities and hence need changes to the kernel. We use a scheme similar to the mitigation scheme where the driver ping-pongs between transmission and reception. Interrupts are turned off during this period of activity. When activity ceases, interrupts are re-enabled. We thus amortize the cost of interrupts and domain switches incurred in the data path.

The “fail fast” concept has recently been rediscovered by Candea *et al.* [CF03], but was well known to earlier practitioners [Gra86, SS83]. Fail-fast (or “crash-only,” as Candea *et al.* call it) techniques were used in production in the KeyKOS operating system (the predecessor to EROS) by 1983 [Har85], and in the KeyTXF transaction processing system by 1987. The notion has been an underlying design pattern in EROS since 1990.

7 Acknowledgments

The network stack in both the monolithic and the factored designs is based on the lwIP TCP/IP protocol suite. Although lwIP was designed primarily for embedded system, we adapted it to EROS because of its small simple code base. The packet demultiplexer used in the factored version of the network subsystem is a simplified version of the soft LRP demultiplexer.

Chandramohan Thekkath graciously permitted us to reproduce the performance results shown in Table 1. Our shepherd, Vivek Pai, was exceptionally patient with us, which allowed us to present our results more accurately.

We would like to thank John Vanderburgh and Eric Northup of the Systems Research Laboratory, Johns Hopkins University for (respectively) helping in the implementation of shared memory in EROS and assisting in debugging the adapted lwIP stacks.

8 Conclusion

We have presented the design and implementation of a domain factored Network Subsystem which provides a defensible multi-layered network subsystem with microbenchmark performance comparable to the existing in-kernel monolithic networking subsystem.

Although a great deal of research has gone into improving the network subsystem, we are aware of no prior work attempting to address both performance and security together, nor work that has successfully factored a network stack into multiple protection domains. In this work, we have demonstrated that such factoring is practical. Our results show that speed need not be sacrificed in any substantial measure to attain security. This suggests that the much-promoted “cost of protection” have been greatly overestimated in previous work. The cache performance results presented here suggest that the cache effects resulting from factoring are more complex than had previously been assumed.

One key to the performance we have achieved is integrating the packet filter concept with scatter-gather technology. The EROS-factored packet filter simultaneously demultiplexes the packets to the appropriate recipient and divides the packets into component parts that are delivered into the appropriate memory region.

When combined with appropriate use of CPU scheduling, the EROS-factored networking stack satisfies all of

the goals identified in Section 2. Fault resilience, protocol flexibility, resource accountability, and performance isolation are achieved without unduly compromising performance. This is possible in part because the EROS kernel interface exposes low-level resources using a protection model that allows us to correctly align the protection mechanisms with the interests of the various parties. In particular, the ability to separate authority to deallocate storage from authority to read that storage is essential to maintaining the trust and resource relationships between the different components.

References

- [AD00] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.
- [BC00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Press, October 2000.
- [BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.
- [Bla96] Trevor Blackwell. Speeding up protocols for small messages. In *Proc. ACM SIGCOMM '96*, pages 85–95, September 1996.
- [CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proc. 14th Symposium on Operating Systems Principles*, December 1993.
- [CF03] George Candea and Armando Fox. Crash-only software. In *Proc. HotOS-IX*, Lihue, Hawaii, USA, May 2003.
- [CPM⁺98] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.
- [DB96] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Operating Systems Design and Implementation*, pages 261–275, 1996.
- [DP93] Peter Druschel and Larry Peterson. Fbufs: A high-bandwidth cross-domain transfer utility. In *Proc. 14th Symposium on Operating Systems Principles*, December 1993.
- [Dun02] Adam Dunkels. lwip - a lightweight tcp/ip stack, October 2002. <http://www.sics.se/~adam/lwip/>.
- [DWB⁺93] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: Architectural support for high-performance protocols, 1993.
- [EK92] D. Engler and M. F. Kaashoek. Dpf: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. SIGCOMM '96 Conference*, pages 53–59, Stanford, CA, USA, August 1992.
- [EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symposium on Operating Systems Principles*, pages 251–266, December 1995.
- [EWL⁺94] Aled Edwards, Greg Watson, John Lumley, David Banks, Costas Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s lan. *SIGCOMM Comput. Commun. Rev.*, 24(4):14–23, 1994.
- [FL94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating threads model. In *Proc. Winter USENIX Conference*, pages 97–114, January 1994.
- [GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. UNIX as an application program. In *Proc. USENIX Summer Conference*, pages 87–96, June 1990.
- [Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, January 1986.
- [Har85] Norman Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, October 1985.

- [HHL⁺97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian. Schönberg, and Jean Wolter. The performance of μ -kernel-based systems. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France., October 1997.
- [KMY01] Ilhwan Kim, Jungwhan Moon, and Heon Y. Yeom. Timer-based interrupt mitigation for high performance packet processing. In *Proc. 5th International Conference on High-Performance Computing in the Asia-Pacific Region*, Gold Coast, Australia, September 2001.
- [Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proc. 14th ACM Symposium on Operating System Principles*, pages 175–188. ACM, 1993.
- [Lie95] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.
- [Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.
- [MPBO96] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean W. O’Malley. Analysis of techniques to improve protocol processing latency. In *SIGCOMM*, pages 73–84, 1996.
- [MRA87] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, USA, November 1987.
- [PDP94] Bruce S. Davie P Druschel and Larry L. Peterson. Experiences with a high-speed network adaptor: A software perspective. In *Proc. of ACM SIGCOMM 94*, 1994.
- [PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: A unified i/o buffering and caching system. In *Proc. Third USENIX Symposium on Operating Systems Design and Implementation*, pages 22–25, New Orleans, Louisiana, USA, February 1999. USENIX Association.
- [SO01] Jamal Hadi Salim and Robert Olsson. Beyond softnet, 2001.
- [SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.
- [SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.
- [ST93] Jonathan M. Smith and C. Brendan S. Traw. Giving applications access to gb/s networking. *IEEE Network*, 7(4):44–52, 1993.
- [Ten01] D.L. Tennenhouse. Layered multiplexing considered harmful, 2001.
- [TNML93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, 1993.
- [vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, 1995.

accept()able Strategies for Improving Web Server Performance

Tim Brecht, David Pariag, Louay Gammo*

School of Computer Science

University of Waterloo

`{brecht,db2pariag,lgamma}@cs.uwaterloo.ca`

Abstract

This paper evaluates techniques for improving the performance of three architecturally different web servers. We study strategies for effectively accepting incoming connections under conditions of high load. Our experimental evaluation shows that the method used to accept new connection requests can significantly impact server performance. By modifying each server's accept strategy, we improve the performance of the kernel-mode TUX server, the multi-threaded Knot server and the event-driven μ server. Under two different workloads, we improve the throughput of these servers by as much as 19% – 36% for TUX, 0% – 32% for Knot, and 39% – 71% for the μ server. Interestingly, the performance improvements realized by the user-mode μ server allow it to obtain performance that rivals an unmodified TUX server.

1 Introduction

Internet-based applications have experienced incredible growth in recent years and all indications are that such applications will continue to grow in number and importance. Operating system support for such applications is the subject of much activity in the research community, where it is commonly believed that existing implementations and interfaces are ill-suited to network-centric applications [4] [30] [23].

In many systems, once client demand exceeds the server's capacity the throughput of the server degrades sharply, and may even approach zero. This is reflected in long (and unpredictable) client wait times, or even a complete lack of response for some clients. Ironically, it is precisely during these periods of high demand that quality of service matters most. Breaking news, changes in the stock market, and even the Christmas shopping season can generate flash crowds or even prolonged periods of overload. Unfortunately, over-provisioning of server capacity is neither cost effective nor

practical since peak demands can be several hundred times higher than average demands [1].

Because modern Internet servers multiplex among large numbers of simultaneous connections, much research has investigated modifying operating system mechanisms and interfaces to efficiently obtain and process network I/O events [3] [4] [22] [23] [7]. Other research [20], has analyzed the strengths and weaknesses of different server architectures. These include multi-threaded (MT), multi-process (MP), single process event-driven (SPED) and even a hybrid design called asymmetric multi-process event-driven (AMPED) architecture. More recent work [31] [9] [29] [28] has re-ignited the debate regarding whether to multiplex connections using threads or events in high-performance Internet servers. In addition, an interesting debate has emerged concerning the relative merits of kernel-mode versus user-mode servers, with some research [15] indicating that kernel-mode servers enjoy significant performance advantages over their user-mode counterparts.

In this paper, we examine different strategies for accepting new connections under high load conditions. We consider three architecturally different web servers: the kernel-mode TUX server [24] [16], the event-driven, user-mode μ server [6] [11], and the multi-threaded, user-mode Knot server [29] [28].

We examine the connection-accepting strategy used by each server, and propose modifications that permit us to tune each server's strategy. We implement our modifications and evaluate them experimentally using workloads that generate true overload conditions. Our experiments demonstrate that accept strategies can significantly impact server throughput, and must be considered when comparing different server architectures.

Our experiments show that:

- Under high loads a server must ensure that it is able to accept new connections at a sufficiently high rate.
- In addition to accepting new connections at a high rate, the server must spend enough time servicing existing connections. That is, a balance must be maintained be-

*Some of the research for this paper was conducted while this author was employed by Hewlett Packard Labs.

tween accepting new connections and working on existing connections.

- Each server that we examine can significantly improve its throughput by improving the aforementioned balance.
- Contrary to previous findings, we demonstrate that a user-level server is able to serve an in-memory, static, SPECweb99-like workload at a rate that compares very favourably with the kernel-mode TUX server.

2 Background and Related Work

Current approaches to implementing high-performance Internet servers require special techniques for dealing with high levels of concurrency. This point is illustrated by first considering the logical steps taken by a web server to handle a single client request, as shown in Figure 1.

1. Wait for and accept an incoming network connection.
2. Read the incoming request from the network.
3. Parse the request.
4. For static requests, check the cache and possibly open and read the file.
5. For dynamic requests, compute the result.
6. Send the reply to the requesting client.
7. Close the network connection.

Figure 1: *Logical steps required to process a client request.*

Note that almost all Internet servers and services follow similar steps. For simplicity, the example in Figure 1 does not handle persistent or pipelined connections (although all servers used in our experiments handle persistent connections).

Many of these steps can block because of network or disk I/O, or because the web server must interact with another process. Consequently, a high performance server must be able to concurrently process partially completed connections by quickly identifying those connections that are ready to be serviced (i.e., those for which the application would not have to block). This means the server must be able to efficiently multiplex several thousand simultaneous connections [4] and to dispatch network I/O events at high rates.

Research into improving web server performance tends to focus on improving operating system support for web servers, or on improving the server's architecture and design. We now briefly describe related work in these areas.

2.1 Operating System Improvements

Significant research [3] [2] [4] [19] [22] [23] [7] has been conducted into improving web server performance by improving both operating system mechanisms and interfaces for obtaining information about the state of socket and file descriptors. These studies have been motivated by the overhead

incurred by `select`, `poll`, and similar system calls under high loads. As a result, much research has focused on developing improvements to `select`, `poll` and `sigwaitinfo` by reducing the amount of data that needs to be copied between user space and kernel space or by reducing the amount of work that must be done in the kernel (e.g., by only delivering one signal per descriptor in the case of `sigwaitinfo`). Other work [21] has focused on reducing data copying costs by providing a unified buffering and caching system.

In contrast to previous research on improving the operating system, this paper presents strategies for accepting new connections which improve server performance under existing operating systems, and which are relevant to both user-mode and kernel-mode servers.

2.2 Server Application Architecture

One approach to multiplexing a large number of connections is to use a SPED architecture, which uses a single process in conjunction with non-blocking socket I/O and an event notification mechanism such as `select` to deliver high throughput, especially on in-memory workloads [20]. The event notification mechanism is used to determine when a network-related system call can be made without blocking. This allows the server to focus on those connections that can be serviced without blocking its single process.

Of course, a single process cannot leverage the processing power of multiple processors. However, in multiprocessor environments multiple copies of a SPED server can be used to obtain excellent performance [32].

The multi-process (MP) and multi-threaded (MT) models [20] offer an alternative approach to multiplexing simultaneous connections by utilizing a thread (or process) per TCP connection. In this approach, connections are multiplexed by context-switching from a thread that can no longer process its connection because it will block, to another thread that can process its connection without blocking. Unfortunately threads and processes can consume large amounts of resources and architects of early systems found it necessary to restrict the number of executing threads [13] [4].

The Flash server implements a hybrid of the SPED and MP models called AMPED (asymmetric multi-process event-driven) architecture [20]. This architecture builds on the SPED model by using several helper processes to perform disk accesses on behalf of the main event-driven process. This approach performed very well on a variety of workloads and outperformed the MP and MT models.

More recent work has revived the debate concerning event-driven versus multi-threaded architectures. Some papers [31] [9] [32] conclude that event-driven architectures afford higher-performance. Others [28] [29] argue that highly efficient implementations of threading libraries allow high performance while providing a simpler programming model.

Our work in this paper uses servers that are implemented

using both event-driven and multi-threaded architectures. We demonstrate that improved accept strategies can increase throughput in either type of server.

2.3 Kernel-mode Servers

In light of the considerable demands placed on the operating system by web servers, some people [24] [12] have argued that the web server should be implemented in the kernel as an operating system service. Recent work [15] has found that there is a significant gap in performance between kernel-mode and user-mode servers when serving memory-based workloads. Our findings in this paper challenge these results. In fact, on a static, in-memory, SPECweb99-like workload the μ server's performance compares very favourably with that of the kernel-mode TUX server.

2.4 Accept Strategies

In early web server implementations, the strategy for accepting new connections was to accept one connection each time the server obtained notification of pending connection requests. Recent work by Chandra and Mosberger [7] discovered that a simple modification to a `select`-based web-server (with a stock operating system) outperformed operating system modifications they and other researchers [22] had performed in order to improve event dispatch scalability. They referred to this server as a *multi-accept* server. Upon learning of a pending connection, this server attempts to accept as many incoming connections as possible by repeatedly calling `accept` until the call fails (and the `errno` is set to `EWOULDBLOCK`) or the limit on the maximum number of open connections is reached. This multi-accept behaviour means that the server periodically attempts to drain the entire accept queue. Their experiments demonstrate that this aggressive strategy towards accepting new connections improved event dispatch scalability for workloads that request a single one byte file or a single 6 KB file.

In this paper, we explore more representative workloads and demonstrate that their multi-accept approach can overemphasize the accepting of new connections while neglecting the processing of existing connections. The resulting imbalance leads to poor performance.

We devise a simple mechanism to permit us to implement and tune a variety of accept strategies, and to experimentally evaluate the impact of different accept strategies on three server architectures. We demonstrate that a carefully tuned accept policy can significantly improve performance across all three server architectures.

More recent work [28] [29] has also noted that the strategy used to accept new connections can significantly impact performance. Our work specifically examines different strategies applied to a variety of servers in order to understand how to choose a good accept strategy.

3 Improving Accept Strategies

In order for a client to send a request to the server it must first establish a TCP connection to the server. This is done by using the TCP three-way handshake [26]. Once the three-way handshake succeeds the kernel adds a socket to the *accept queue* (sometimes referred to as the listen queue) [5]. Each time the server invokes the `accept` system call a socket is removed from the front of the accept queue, and an associated file descriptor is returned to the server.

In Linux, the length of the accept queue is theoretically determined by the application when it specifies a value for the `backlog` parameter to the `listen` system call. In practice however, the Linux kernel silently limits the `backlog` parameter to a maximum of 128 connections. This behaviour has been verified by examining several Linux kernel versions (including 2.4.20-8 and 2.6.0-test7). In our work, we have intentionally left this behaviour unchanged because of the large number of installations that currently operate with this limit.

If the server accepts new connections too slowly, then either the accept queue or the SYN queue will quickly fill up. If either queue fills, all new connection requests will be dropped. Such queue drops are problematic for both client and server. The client is unable to send requests to the server, and is forced to re-attempt the connection. Meanwhile, the server-side kernel has invested resources to process packets and complete the TCP three-way handshake, only to discover that the connection must be dropped. For these reasons, queue drops should be avoided whenever possible.

Our work in this paper concentrates on improving accept strategies to enable servers to accept and process more connections. Note that this is quite different from simply reducing the number of queue drops (i.e., failed connection attempts) because queue drops could be minimized by only ever accepting connections and never actually processing any requests. Naturally this alone would not lead to good performance. Instead our strategies focus on finding a balance between accepting new connections and processing existing connections.

4 The Web Servers

This section describes the architecture of each of the servers investigated as well as the procedure each uses for accepting new connections. We also describe any modifications we have made to the base server behaviour.

4.1 The μ server

The micro-server (μ server) [6] is a single process event-driven web server. Its behaviour can be carefully controlled through the use of a large number of command-line parameters, which allow us to investigate the effects of several different server configurations using a single web-server. The

`μserver` uses either the `select`, `poll`, or `epoll` system call (chosen through command line options) in concert with non-blocking socket I/O to process multiple connections concurrently.

The `μserver` operates by tracking the state of each active connection (states roughly correspond to the steps in Figure 1). It repeatedly loops over three phases. The first phase (which we call the *getevents-phase*) determines which of the connections have accrued events of interest. In our experiments this is done using `select`. The second phase (called the *accept-phase*) is entered if `select` reports that connections are pending on the listening socket. The third phase (called the *work-phase*) iterates over each of the non-listening connections that have events of interest that can be processed without blocking. Based on the event-type and the state of the connection, the server calls the appropriate function to perform the required work. A key point is that for the `μserver` options used in our experiments the work-phase does not consider any of the new connections accumulated in the immediately preceding accept-phase. That is, it only works on connections when `select` informs it that work can proceed on that connection without blocking.

The `μserver` is based on the multi-accept server written by Chandra and Mosberger [7]. That server implements an accept policy that drains its accept queue when it is notified of a pending connection request. In contrast, the `μserver` uses a parameter that permits us to accept up to a pre-defined number of the currently pending connections. This defines an upper limit on the number of connections accepted consecutively. For ease of reference, we call this parameter the accept-limit parameter, and refer to it throughout the rest of this paper. The same name is also used to refer to similar modifications to Knot and TUX. Parameter values range from one to infinity (*Inf*). An accept-limit of one forces the server to accept a single connection in each accept-phase, while *Inf* causes the server to accept all currently pending connections.

Our early investigations [6] revealed that the accept-limit parameter could significantly impact the `μserver`'s performance. This motivated us to explore the possibility of improving the performance of other servers, as well as quantifying the performance gains under more representative workloads. As a result, we have implemented accept-limit mechanisms in two other well-known web servers. We now describe these servers and their accept mechanisms.

4.2 Knot

Knot [28] is a multi-threaded web server which makes use of the Capriccio [29] threading package. Knot is a simple web server. It derives many benefits from the Capriccio threading package, which provides lightweight, cooperatively scheduled, user-level threads. Capriccio features a number of different thread schedulers, including a resource-aware scheduler which adapts its scheduling policies according to the ap-

plication's resource usage. Knot operates in one of two modes [28] which are referred to as Knot-C and Knot-A.

Knot-C uses a thread-per-connection model, in which the number of threads is fixed at runtime (via a command-line parameter). Threads are pre-forked during initialization. Thereafter, each thread executes a loop in which it accepts a single connection and processes it to completion. Knot-A creates a single *acceptor* thread which loops attempting to accept new connections. For each connection that is accepted, a new *worker* thread is created to completely process that connection.

Knot-C is meant to favour the processing of existing connections over the accepting of new connections, while Knot-A is designed to favour the accepting of new connections. By having a fixed number of threads and using one thread per connection, Knot-C contains a built-in mechanism for limiting the number of concurrent connections in the server. In contrast, Knot-A allows increased concurrency by placing no limit on the number of concurrent threads or connections.

Our preliminary experiments revealed that Knot-C performs significantly better than Knot-A, especially under overload where the number of threads (and open connections) in Knot-A becomes very large. These results agree with findings reported by the authors of Knot [28], and as a result we focus our tuning efforts on Knot-C.

We modified Knot-C to allow each of its threads to accept multiple connections before processing any of the new connections. This was done by implementing a new function that is a modified version of the accept call in the Capriccio library. This call loops to accept up to accept-limit new connections provided that they can be accepted without blocking. If the call to accept would block and at least one connection has been accepted the call returns and the processing of the accepted connections proceeds. Otherwise the thread is put to sleep until a new connection request arrives. After accepting new connections, each thread fully processes the accepted connections before admitting any new connections. Therefore, in our modified version of Knot each thread oscillates between an accept-phase and a work-phase. As in the `μserver`, the accept-limit parameter ranges from 1 to infinity. The rest of this paper uses the accept-limit parameter to explore the performance of our modified version of Knot-C. Note that when the accept-limit is set to 1 our modified version of Knot operates in the same fashion as the original.

4.3 TUX

TUX [24] [16] is an event-driven kernel-mode web server for Linux developed by Red Hat. It is compiled as a kernel-loadable module (similar to many Linux device drivers), which can be loaded and unloaded on demand. TUX's kernel-mode status affords it many I/O advantages including true zero-copy disk reads, zero-copy network writes, and zero copy request parsing. In addition, TUX accesses kernel data

structures (e.g., the listening socket's accept queue) directly, which allows it to obtain events of interest with relatively low overhead when compared to user-level mechanisms like `select`. Lastly, TUX avoids the overhead of kernel cross-calls that user-mode servers must incur when making system calls. This is important in light of the large number of system calls needed to process a single HTTP request.

A look at the TUX source code provides detailed insight into TUX's structure. TUX's processing revolves around two queues. The first queue is the listening socket's accept queue. The second is the `work.pending` queue which contains items of work (e.g., reads and writes) that are ready to be processed without blocking. TUX oscillates between an accept-phase and a work-phase. It does not require a getevents-phase because it has access to the kernel data structures where event information is available. In the accept-phase TUX enters a loop in which it accepts all pending connections (thus draining its accept queue). In the work-phase TUX processes all items in the `work.pending` queue before starting the next accept-phase. Note that new events can be added to each queue while TUX removes and processes them.

We modified TUX to include an accept-limit parameter, which governs the number of connections that TUX will accept consecutively before leaving the accept-phase. Since TUX is a kernel-loadable module, it does not accept traditional command line parameters. Instead, the new parameter was added to the Linux `/proc` file system, in the `/proc/sys/net/tux` subdirectory. The `/proc` mechanism is convenient in that it allows the new parameter to be read and written without restarting TUX. This parameter gives us a measure of control over TUX's accept policy, and allows us to compare different accept-limit values with the default policy of accepting all pending connections.

Note that there is an important difference between how the `μserver` and TUX operate. In the `μserver` the work-phase processes a fixed number of connections (determined by `select`). In contrast TUX's `work.pending` queue can grow during processing, which prolongs its work phase. As a result we find that the accept-limit parameter impacts these two servers in dramatically different ways. This will be seen and discussed in more detail in Section 6.

It is also important to understand that the accept-limit parameter does not control the accept rate, but merely influences it. The accept rate is determined by a combination of the frequency with which the server enters the accept-phase and the number connections accepted while in that phase. The amount of time spent in the work and getevent phases determines the frequency with which the accept-phase is entered.

5 Experimental Methodology

In our graphs, each data point is the result of a two minute experiment. Trial and error revealed that two minutes provided

sufficient time for each server to achieve steady state execution. Longer durations did not alter the measured results, and only served to prolong experimental runs. A two minute delay was used between consecutive experiments. This allowed all TCP sockets to clear the `TIME_WAIT` state before commencing the next experiment. Prior to running experiments, all non-essential Linux services (e.g., `sendmail`, `dhcpd`, `cron` etc.) were shutdown. This eliminated interference from daemons and periodic processes (e.g., `cron jobs`) which might confound results.

Prior to determining which accept-limit values to include in each graph a number of alternatives were run and examined. The final values presented in each graph were chosen in order to highlight the interesting accept policies and differences. The following sections describe our experimental environment and the parameters used to configure each server.

5.1 Environment

Our experimental environment is made up of two separate client-server clusters. The first cluster (Cluster 1) contains a single server and eight clients. The server contains two Xeon processors running at 2.4 GHz, 1 GB of RAM, a 10,000 RPM SCSI disk, and two Intel e1000 Gbps Ethernet cards. The clients are identical to the server with the exception of their disks which are EIDE. The server and clients are connected with a 24-port Gbps switch. Since the server has two cards, we avoid network bottlenecks by partitioning the clients into different subnets. In particular, the first four clients communicate with the server's first ethernet card, while the remaining four use a different IP address linked to the second ethernet card.

Each client runs Red Hat 9.0 which uses the 2.4.20-8 Linux kernel. The server also uses the 2.4.20-8 kernel, but not the binary that is distributed by Red Hat. Instead, the Red Hat sources were re-compiled after we incorporated our changes to TUX. The resulting kernel was used for all experiments on this machine. The aforementioned kernel is a uni-processor kernel that does not provide SMP support. The reasons for this are twofold. Firstly, the Capriccio threading package does not currently include SMP support. Secondly, we find it instructive to study the simpler single-processor problem, before considering complex SMP interactions.

The second machine cluster (Cluster 2) also consists of a single server and eight clients. The server contains two Xeon processors running at 2.4 GHz, 4 GB of RAM, several high-speed SCSI drives and two Intel e1000 Gbps Ethernet cards. The clients are dual-processor Pentium III machines running at 550 MHz. Each client has 256 MB of RAM, an EIDE disk, and one Intel e1000 Gbps Ethernet card. The server runs a Linux 2.4.19 uni-processor kernel, while the clients use the 2.4.7-10 kernel that ships with Red Hat 7.1.

This cluster of machines is networked using a separate 24-port Gbps switch. Like the first cluster, the clients are di-

vided into two groups of four with each group communicating with a different server NIC. In addition to the Gbps network, all machines are connected by a separate 100 Mbps network which is used for co-ordinating experiments. Each cluster is completely isolated from other network traffic.

Cluster 1 is used to run all μ server and TUX experiments while Cluster 2 is used to run all Knot experiments. Because our clusters are slightly different, we do not directly compare results taken from different clusters. Instead, each graph presents data gathered from a single cluster. Ideally, we would use one cluster for all our experiments, but the number of experiments required necessitated the use of two clusters.

5.2 Web Server Configuration

In the interest of making fair and scientific comparisons, we carefully configured TUX and the μ server to use the same resource limits. TUX was configured to use a single kernel thread. This enables comparisons with the single process μ server, and was also recommended in the TUX user manual [24]. The TUX accept queue backlog was set to 128 (via the `/proc/sys/net/tux/max_backlog` parameter) which matches the value imposed on the user-mode servers. By default, TUX bypasses the kernel-imposed limit on the length of the accept queue, in favour of a much larger backlog (2,048 pending connections). This adjustment also eases comparison and understanding of accept-limit-Inf strategies.

Additionally, both TUX and the μ server use limits of 15,000 simultaneous connections. In the μ server case this is done by using an appropriately large `FD_SETSIZE`. For TUX this was done through `/proc/sys/net/tux/max_connections`. All μ server and TUX experiments were conducted using the same kernel.

The Knot server was configured to use the Knot-C behaviour. That is, it pre-forks and uses a pre-specified number of threads. In our case we used 1,000 threads. Although we have not extensively tuned Knot we have noticed that as long as the number of threads was not excessively small or large, the performance of Knot-C was not greatly impacted by the number of threads. Note that in this architecture the number of threads used also limits the maximum number of simultaneous connections. When the accept-limit modification is added to Knot it permits several connections per thread to be open, thus increasing this limit.

Finally, logging is disabled on all servers and we ensure that all servers can cache the entire file set. This ensures that differences in server performance are not due to caching strategies.

6 Workloads and Experimental Results

This section describes the two workloads used in our experiments, and discusses the results obtained with each of the

three servers. Our results show that the accept strategy significantly impacts server performance for each server.

6.1 SPECweb99-like Workload

The SPECweb99 benchmarking suite [25] is a widely accepted tool for evaluating web server performance. However, the suite is not without its flaws. The SPECweb99 load generators are unable to generate loads that exceed the capacity of the server. The problem is that the SPECweb99 load generator will only send a new request once the server has replied to its previous request. Banga et al. [5] show that under this approach the clients' request rates are throttled by the server. As such, the clients are unable to overload the server.

We address this problem by using *httperf*, an http load generator that is capable of generating overload [17]. *httperf* avoids the naive load generation scheme by implementing connection timeouts. Every time a connection to the server is initiated, a timer is started. If the connection timer expires before the connection is established and the HTTP transaction completes, the connection is aborted and retried. This allows the clients to generate loads that exceed the server's capacity. We use *httperf* in conjunction with a SPECweb99 file set and a session log file that we have constructed to mimic the SPECweb99 workload. Although our traces are synthetic, they are carefully generated to accurately recreate the file classes, access patterns, and the number of requests issued per persistent HTTP 1.1 connection used in the static portion of SPECweb99 [25].

In all experiments, the SPECweb99 file set and server caches are sized so that the entire file set fits in main memory. This is done to eliminate differences between servers due to differences in caching implementations. While an in-memory workload is not entirely representative, it does permit us to compare our results with those of Joubert et al. [15], who used an in-memory SPECweb96 workload to compare the performance of kernel-mode and user-mode servers.

Figure 2 examines the performance of the μ server as the accept-limit parameter is varied. Recall that the accept-limit parameter controls the number of connections that are accepted consecutively. This graph shows that a larger accept-limit can significantly improve performance in the μ server, especially under overload. In fact, at the extreme target load of 30,000 requests/sec, the accept-limit-Inf policy outperforms the accept-limit-1 policy by 39%.

Statistics collected by the μ server provide insights that confirm the benefits of the high accept-limit value. At a target load of 30,000 requests/sec, the accept-limit-Inf server accepts an average of 1,571 new connections per second. In comparison, the accept-limit-1 server averages only 1,127 new connections per second (28% fewer). This difference is especially significant when we consider that each SPECweb99 connection is used to send an average of 7.2 requests. Figure 3 shows that in all cases the higher accept-rates

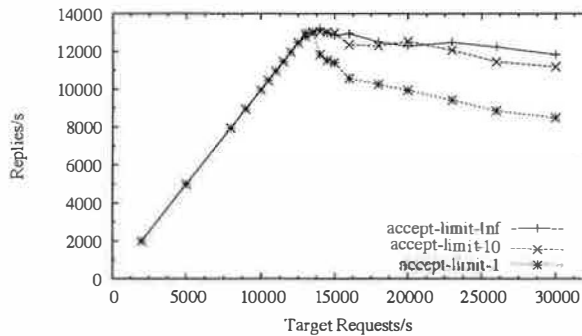


Figure 2: μ server performance under SPECweb99-like workload

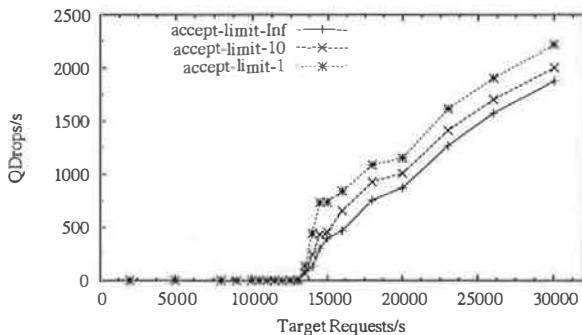


Figure 3: μ server queue drops/sec under SPECweb99-like workload

result in lower queue drop rates (QDrops/s). The lower drop rates mean that less time is wasted in the processing of packets that will be discarded, and more time can be devoted to processing client requests. As seen in Figure 2, this translates into a healthy improvement in throughput.

The queue drop rates are obtained by running *netstat* on the server before and after each experiment. The number of *failed TCP connection attempts* and *listen queue overflows* is summed and recorded before and after each experiment. Subtracting these values and dividing by the experiment's duration provides a rate, which we report in our queue drop graphs.

For the Knot server, we experimented with a variety of different accept strategies. The results are summarized in Figures 4 and 5. Figure 4 illustrates the throughput obtained using different accept policies. With the *accept-limit* parameter set to 1, our modified version of Knot behaves identically to an unmodified copy of Knot. As a sanity check, we confirmed that the original version and the modified server using the *accept-limit-1* policy produce indistinguishable results. To reduce clutter, we omit results for the original version of Knot.

Higher *accept-limits* (10, 50 and 100) represent our attempts to increase Knot's throughput by increasing its accept rate. Our server-side measurements confirm that we are able to increase Knot's accept rate. For example, statistics col-

lected in Knot reported that at a load of 20,000 requests/sec, the *accept-limit-100* policy accepts new connections 240% faster (on average) than the *accept-limit-1* (default) server. Further evidence is provided in Figure 5 which shows that the *accept-limit-50* and *accept-limit-100* servers enjoy significantly lower queue drop rates than their less aggressive counterparts.

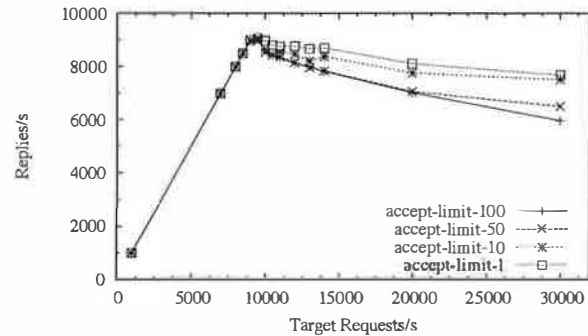


Figure 4: Knot performance under SPECweb99-like workload

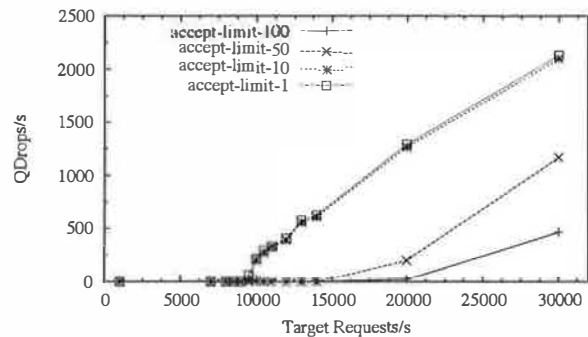


Figure 5: Knot queue drops/sec under SPECweb99-like workload

Unfortunately, the higher accept rates (and lowered queue drop rates) do not improve performance. On the contrary, performance suffers. Statistics reported by Knot show that with an *accept-limit* of 50 or higher, the number of concurrent connections in the server grows quite sharply. We believe that performance degrades with a large number of connections because of overheads in the Capriccio threading library. As a result, we find that under this workload, more aggressively accepting new connections does not improve Knot's performance. These findings agree with previously published results [28] in which overly aggressive accepting also hurt Knot's performance.

In Figure 6 we show that the *accept-limit* parameter can be used to improve TUX's performance. The *accept-limit-Inf* policy corresponds to TUX's default accept behaviour (draining the accept queue). The *accept-limit-50* policy allows TUX to consecutively accept up to 50 connections, while the *accept-limit-1* policy limits TUX to accepting a single con-

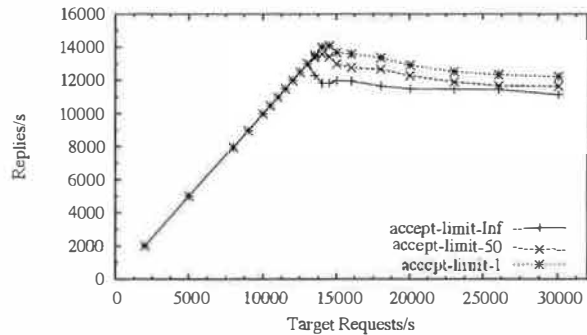


Figure 6: TUX performance under SPECweb99-like workload

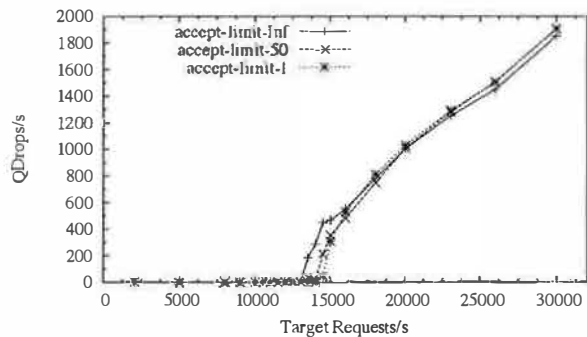


Figure 7: TUX queue drops/sec under SPECweb99-like workload

nection in each accept-phase. Figure 6 shows that the accept-limit-1 policy results in a 12% increase in peak throughput, and a 19% increase in throughput at 14,500 reqs/sec. Surprisingly, our server-side instrumentation shows that an accept-limit-1 policy causes TUX to accept connections *faster* than the higher accept-limit values. While this behaviour may seem unintuitive, it is important to remember that TUX's accept rate is not directly governed by the accept-limit parameter. Rather, the accept-limit controls the maximum number of connections that are accepted consecutively. The server's accept rate is determined by the number of consecutive accepts as well as the number of times that TUX enters its accept-phase.

Equation 1 formalizes this simple mathematical relationship. In this equation, $t_{elapsed}$ denotes the elapsed time for a given experiment, N_{phases} represents the number of accept-phases the server completes during the experiment, and C_{avg} denotes the average number of new connections accepted per accept-phase. In our experiments, $t_{elapsed}$ is essentially a constant.

$$AcceptRate = \frac{N_{phases} C_{avg}}{t_{elapsed}} \quad (1)$$

In TUX, lowering the accept-limit has two effects. Firstly, C_{avg} decreases since each accept-phase is shortened. Secondly, N_{phases} increases dramatically. In our experiments,

the increase in N_{phases} outweighs the decrease in C_{avg} and leads to a net increase in the observed accept rate. We found that for low accept-limits, TUX accepted fewer connections in each accept-phase, but entered its accept-phase more frequently (because the low accept-limit shortened its work-phase). On balance, lower accept-limits lead to a higher accept rate.

Interestingly, the accept-limit parameter has a very different effect on TUX and the μ server in spite of the fact that both are event-driven servers with accept-phases and work-phases. Because of this similarity, Equation (1) applies equally to both servers. In the μ server, lowering the accept-limit parameter also lowers C_{avg} , and increases N_{phases} . However, in this case the increase in N_{phases} is unable to compensate for the decrease in C_{avg} . As a result, the μ server's accept-rate falls when its accept-limit is lowered.

This analysis shows that in spite of the same *relative* change in N_{phases} and C_{avg} , the *magnitude* of each change is quite different in the μ server and TUX. The difference in magnitude arises because of the get-events phase that exists in the μ server but not in TUX. In the μ server each accept-phase is preceded by a get-events phase (essentially a call to `select`). Increasing the number of accept-phases also increases the number of `select` calls. This adds an overhead to each accept-phase, and limits the μ server's ability to perform more accept-phases. In comparison, TUX incurs no extraneous overhead for extra accept-phases.

Figure 7 shows TUX's queue drop rates for each accept policy. In this case the largest differences in drop rates are seen in the 12,000 to 15,000 requests per second range where there are also the largest differences in reply rates.

6.2 One-packet Workload

In the aftermath of the September 11th 2001 terrorist attacks, many on-line news services were flooded with requests. Many services were rendered unavailable, and even large portals were unable to deal with the deluge for several hours. The staff at CNN.com resorted to replacing their main page with a small, text-only page containing the latest headlines [8]. In fact, CNN sized the replacement page so that it fit entirely in a single TCP/IP packet. This clever strategy was one of the many measures employed by CNN.com to deal with record-breaking levels of traffic.

These events reinforce the need for web servers to efficiently handle requests for small files, especially under extreme loads. With this in mind, we have designed a static workload that tests a web server's ability to handle a barrage of short-lived connections. The workload is simple; all requests are for the same file, issuing one HTTP 1.1 request per connection. The file is carefully sized so that the HTTP headers and the file contents fill a single packet. This resembles the type of requests that would have been seen by CNN.com on September 11.

Obviously, this workload differs from the SPECweb99-like workload in several key respects. For instance, it places much less emphasis on network I/O. Also, because a small file is being requested with each new connection it stresses a server's ability to handle much higher demand for new connection requests. We believe that when studying servers under high loads that this is now an interesting workload in its own right. We also believe that it can provide valuable insights that may not be possible using the SPECweb99-like workload. For more discussion related to the workloads used in this paper see Section 7.

Figure 8 shows the reply rate observed by the clients as the load (target requests per second) on the server increases. This graph shows that the accept-limit-Inf and accept-limit-10 options significantly increase throughput when compared with the naive accept-limit-1 strategy. This is because these servers are significantly more aggressive about accepting new connections than the accept-limit-1 approach. Interestingly, the accept-limit-10 strategy achieves a slightly higher peak than the accept-limit-Inf strategy, although it experiences larger decreases in throughput than accept-limit-Inf as the load increases past saturation. This indicates that better performance might be obtained by dynamically adjusting the accept strategy (this is something we plan to investigate in future research).

The differences in performance between the accept-limit-10 and accept-limit-Inf policies can be seen by examining their ability to accept new connections. Figure 9 shows the queue drop rates for the different accept strategies. Here we see that the μ server operating with an accept-limit of 10 is better able to accept new connections. In fact it is able to avoid significant numbers of queue drops until 23,000 requests per second. On the other hand the accept-limit-Inf option experiences significant numbers of queue drops at 21,500 requests per second. Both of these points correspond to their respective peak rates.

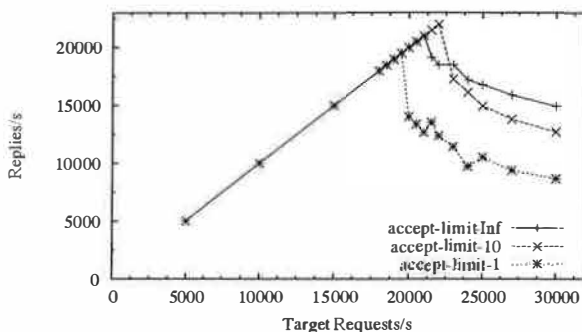


Figure 8: μ server performance under one-packet workload

Figure 9 also shows that the accept-limit-1 option does a good job of accepting new connections until a target request rate of 20,000 requests per second. At that point it is unable to keep up with the demand for new connections. The result

is that the queue drop rate is 17,832 drops per second, and the reply rate is 14,058 replies per second. Significant expense is incurred in handling failed connection requests. If the server can accept those connections, it can improve performance as long as existing connections are not neglected.

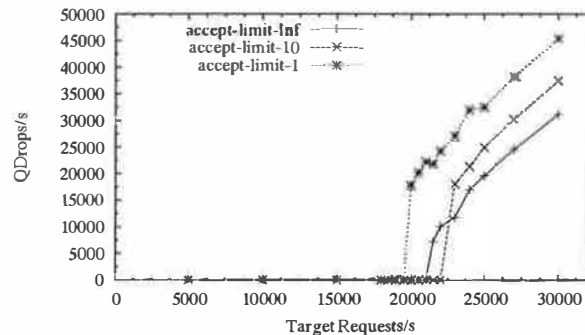


Figure 9: μ server queue drops/sec under one-packet workload

Interestingly, the total of these two rates ($17,832 + 14,058 = 31,890$) exceeds the target request rate of 20,000 requests per second. This is because when a client is attempting to establish a TCP connection using the three-way handshake, if the client does not receive a SYN-ACK packet in response to the SYN packet it sends to the server, it will eventually time-out and retry, which leads to several queue drops per connection.

Using this one-packet workload we are able to increase the μ server's peak throughput from 19,500 replies per second using the naive accept strategy (accept-limit-1) to 22,000 replies per second using the accept-limit-10 strategy. This is an improvement of 13%. More importantly, the accept-limit-Inf strategy improves performance versus the naive strategy by as much as 65% at 21,000 requests per second and 71% at 30,000 requests per second.

Figure 10 shows the reply rate versus the target request rate for the TUX server. As with the SPECweb99-like workload, limiting the number of consecutive accepts increases TUX's accept rate. This can be seen by comparing the queue drop rates (QDrops/sec) in Figure 11 for the different TUX configurations examined. In TUX, the accept-limit-1 strategy does the best job of accepting new connections resulting in the lowest queue drop rates of the configurations examined. This translates directly into the highest throughput.

Recall that the accept-limit-Inf strategy corresponds to the original TUX accept strategy. In this case the improved accept-limit-1 strategy results in a peak reply rate of 22,998 replies per second compared with the original, whose peak is at 20,194 replies per second. This is an improvement of 14%. Additionally there is an improvement of 36% at 23,000 requests per second.

We believe further improvements are possible. However, the simple method we used to modify TUX does not permit

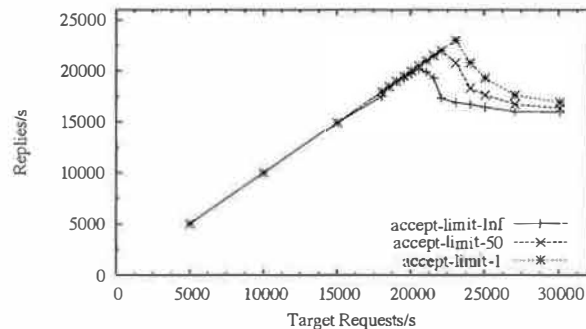


Figure 10: *TUX performance under one-packet workload*

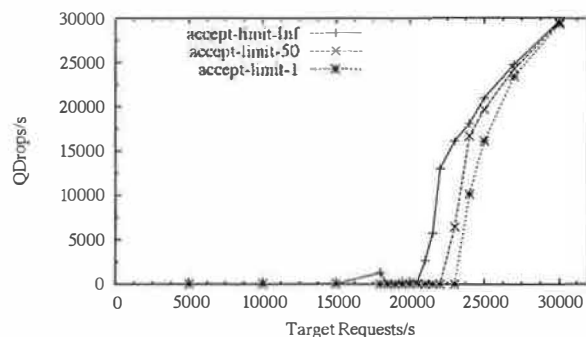


Figure 11: *TUX queue drops/sec under one-packet workload*

us to accept fewer than one connection per accept phase. Ultimately we believe that the best way to control the accept strategy used in TUX, and to control the scheduling of work in general, is to track the number of entries contained in the accept queue and in the number of entries in `work.pending` queue. With this information, a more informed decision can be made about whether to enter an accept-phase or a work-phase. We also believe that limits should be placed on the amount of time spent in each phase, possibly by limiting the number of events processed from each queue. We believe that this approach might be used to further increase the rate at which the server accepts new connections. The difficulty lies in ensuring that the server strikes a balance between accepting new connections and processing existing connections.

For this one packet workload, Knot also benefits from tuning its accept policy. Figure 12 shows an interesting spectrum of accept policies. We observe that the `accept-limit-50` strategy noticeably improves throughput when compared with the original accept strategy. Firstly, peak throughput is increased by 17% from 12,000 to 14,000 replies per second. Secondly, the throughput is increased by 32% at 14,000 requests per second and 24% at 30,000 requests per second.

Interestingly, increasing the `accept-limit` value too much (for example to 100) can result in poor performance. In comparing the `accept-limit-100` strategy with the `accept-limit-1` (default) strategy, we observe that the former obtains a slightly higher peak. However, throughput degrades signifi-

cantly once the saturation point is exceeded. Figure 13 shows how the connection failure rates are impacted by the changes in the accept strategy. Here we see that the `accept-limit-100` version is able to tolerate slightly higher loads than the original before suffering from significant connection failures. The `accept-limit-50` version is slightly better, and in both cases peak throughput improves. At request rates of 15,000 and higher the `accept-limit-50` and `accept-limit-100` strategies do a slightly better job of preventing queue drops than the server using an `accept-limit` of 1. Interestingly, queue drop rates for the `accept-limit 50` and `100` options are quite comparable over this range, yet, there is a large difference in performance. The statistics printed by the Knot server show that at 15,000 requests/sec the `accept-limit-50` policy operates with approximately 25,000 active connections, while the `accept-limit-100` policy is operating with between 44,000 to 48,000 active connections. One possible explanation for the difference in performance is that the overhead incurred by `poll` becomes prohibitive as the number of active connections climbs. These experiments also highlight that a balanced accept policy provides the best performance.

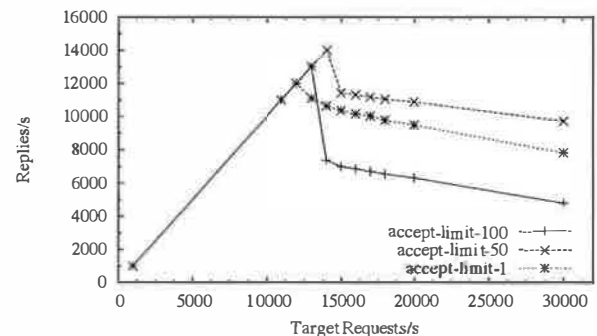


Figure 12: *Knot performance under one-packet workload*

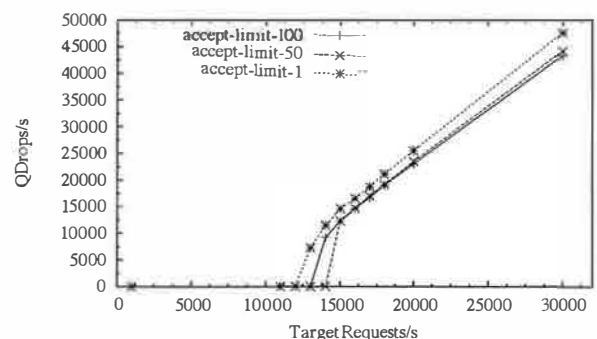


Figure 13: *Knot queue drops/sec under one-packet workload*

6.3 Comparing the μ server and TUX

Figures 14 and 15 compare the performance of the TUX server with the performance of the μ server under the SPECweb99 and one packet workloads respectively. These

graphs show that the original version of TUX (accept-limit-Inf) outperforms a poorly tuned (accept-limit-1) version of the user-mode μ server by as much as 28% under the SPECweb99-like workload and 84% under the one-packet workload (both at 30,000 requests/sec). However, the performance gap is greatly reduced by adjusting the μ server's accept policy. As a result we are able to obtain performance that compares quite favourably with the performance of the unmodified TUX server under both workloads.

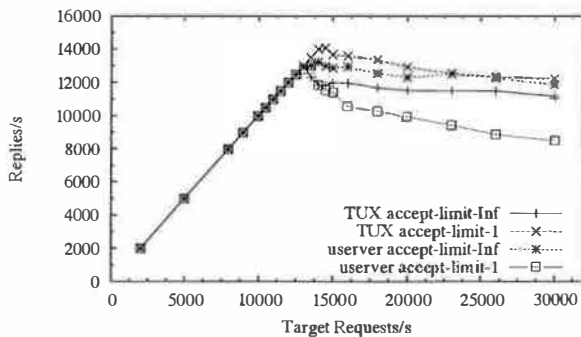


Figure 14: μ server versus TUX performance under SPECweb-like workload

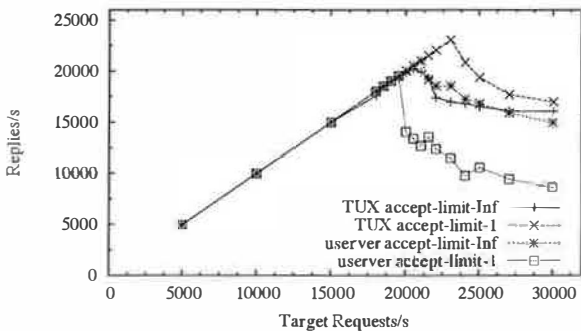


Figure 15: μ server versus TUX performance under one-packet workload

Figure 16 contrasts a superior accept policy with an inferior one for each server. The observed performance differences can be partly explained by examining the queue drop counts for each policy. Under Linux (kernel version 2.4.20-8) a client's connection request may be denied by the server for a variety of reasons (in the order listed), including:

- **SynQFull** : The SYN queue is full when the SYN packet arrives
- **AcceptQFull** : The accept queue is full when the SYN packet arrives
- **DropRequest** : The SYN queue is 3/4 full when the SYN packet arrives
- **ListenOverflow** : The accept queue is full when the SYN-ACK packet arrives

To provide a more complete view of how queue drops impact performance, we added several counters to the Linux kernel. These allow us to categorize queue drops according to the cases outlined above. Queue drop data was obtained by re-running selected experiments under our modified kernel. The throughputs obtained under the modified kernel are comparable to those obtained under the standard kernel. Figure 16 shows detailed queue drop counts for TUX and the μ server under the one packet workload at request rates of 24,000, 27,000 and 30,000 requests/sec. In this figure, the SynQFull count is always zero, and has been omitted.

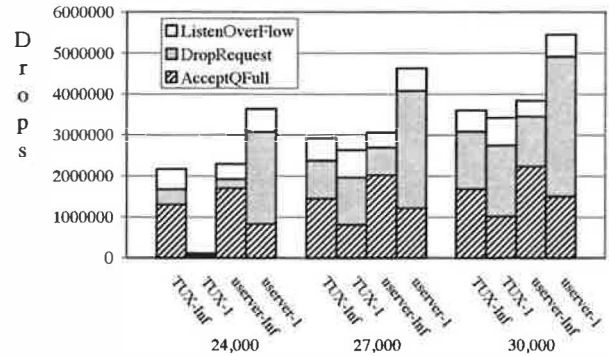


Figure 16: Different types of queue drops for TUX and the μ server under one packet workload for selected request rates

The results for the μ server at 24,000 requests per second show that the high-performance accept-limit-Inf policy (userver-Inf) significantly reduces the number of DropRequests and the overall number of queue drops when compared with the accept-limit-1 policy. The decrease in the number of dropped SYN packets demonstrates that the accept-limit-Inf policy successfully reduces the size of the SYN backlog. However, this reduction is partially offset by a higher AcceptQFull count. The latter can be attributed to the bursty nature of the accept-limit-Inf policy.

By totally draining the accept queue, the accept-limit-Inf policy produces large *accept-phases* where several hundred connections may be accepted consecutively. These long accept phases are followed by long *work-phases* which are needed to process connections. For example, at 24,000 requests/sec the average work-phase under the accept-limit-Inf policy processes 184.9 connections. In comparison, the average work-phase under the accept-limit-1 policy processes just 3.6 connections. During these long work-phases, no new connections are accepted and both the SYN queue and the accept queue accumulate entries. However, it is the much shorter accept queue (128 entries versus 1024 for the SYN queue) that fills first, leading to higher AcceptQFull counts.

The relatively short work-phases of the accept-limit-1 policy also mean that the server does relatively little work per *select* call. As a result, the server must make many more *select* calls to process the same number of connections. The μ server statistics at the request rate of 24,000 indicate

that the `accept-limit-1` policy makes 12,507 `select` calls per second, compared to only 267 calls per second for the `accept-limit-Inf` policy. Clearly, the `accept-limit-1` policy provides a poor amortization of `select` overhead, which hurts performance. The results at 27,000 and 30,000 requests/sec are qualitatively similar.

The results for TUX reveal that at 24,000 requests per second, the `accept-limit-1` policy reduces TUX's queue drops dramatically. This more aggressive accept strategy is successful in keeping both the SYN queue, and the accept queue relatively empty, and the resulting performance is quite good. For TUX, the more aggressive `accept-limit-1` policy mostly reduces `AcceptQFull` counts. TUX with `accept-limit-1` obtains higher accept rates not by accepting more connections in bigger batches (as is done in the μ server) but by more frequently accepting one connection. The result is a less bursty accept policy.

We note that previous research [27] [10] has investigated techniques for reducing the overheads associated with queue drops. We believe such techniques can complement a well-chosen accept strategy. Together, they should provide higher throughputs and more stable overload behaviour.

Figure 17 shows the mean response time for the μ server and TUX using the `accept-limit-1` and `accept-limit-Inf` policies as the load increases. These graphs are obtained by measuring the latency of each completed HTTP 1.1 request at the client under the SPECweb99-like workload.

Although it may be difficult to discern, the graph shows that mean response times are essentially zero at low loads. However, once the servers become saturated the mean response times increase significantly. The μ server with `accept-limit-1` has the lowest mean response times under overload conditions. This is because it does not spend very much time accepting new connections and is able to process the requests that it does accept relatively quickly. In contrast the mean response times for the μ server with `accept-limit-Inf` are higher because the server is quite bursty, and alternates between long accept-phases and long work-phases.

Under overload conditions TUX with `accept-limit-1` obtains both high throughput and relatively low mean response times (in contrast to TUX with `accept-limit-Inf`). With `accept-limit-1`, the accept phases are shorter, permitting work phases to be processed sooner. This translates into a high accept-rate and little burstiness, and provides both high throughput and reasonably low response times.

Our comparison of user-mode and kernel-mode servers produces considerably different results than recent work by Joubert et al. [15]. Their research concludes that kernel-mode servers perform two to three times faster than their user-mode counterparts when serving in-memory workloads. Their experiments on Linux demonstrate that TUX achieved 90% higher performance than the fastest user-mode server (Zeus) measured on Linux. While there are undeniable benefits to the kernel-mode architecture (integration with the TCP/IP

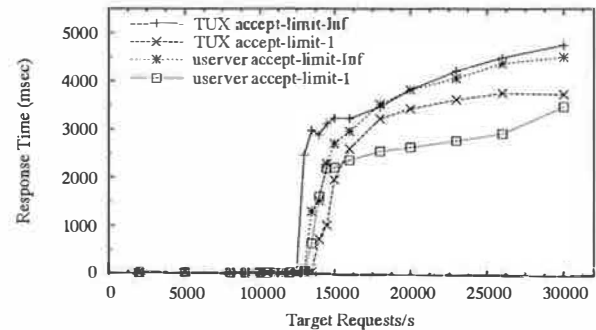


Figure 17: μ server and TUX latencies under the SPECweb99-like workload

stack, zero copy disk I/O, eliminating kernel crossings, etc.), our comparison shows that a user-mode server can rival the performance of TUX.

Some of the gains in user-mode performance are due to the zero-copy `sendfile` implementation that is now available on Linux. There are also differences in workloads. Specifically, Joubert et al. used a HTTP 1.0 based SPECweb96 workload, while we use a HTTP 1.1 based SPECweb99 workload. Lastly, we note the use of different operating system versions, a different performance metric, and possibly different server configurations. In spite of these differences, our work demonstrates that a well tuned user-mode server can closely rival the performance of a kernel-mode server under representative workloads.

7 Discussion

Accept strategies can have considerable impact on web server performance. As a result, we believe these strategies should be considered (along with other parameters that affect performance) when comparing different web servers. We point out that every server has an implicit accept strategy. Perhaps without realizing it, every server makes a decision regarding what portion of the available work should be immediately processed. We emphasize that we have not fully explored the parameter space of possible accept strategies. Instead, we have devised a simple method for demonstrating that accept strategies can have considerable impact on performance in three very different servers. In the future, we plan to investigate techniques for dynamically obtaining a balanced accept strategy that will self-tune for different hardware, operating systems, and even server architectures.

This paper presents a detailed comparison of the μ server and TUX. It is tempting to compare the graphs containing the μ server and Knot results in order to compare the performance of the event-driven and multi-threaded servers. However, such a comparison would be unfair since Knot and the μ server were run in slightly different (hardware and software) environments. As a result, we refrain from direct compar-

isons of these two servers.

The results obtained with the two workloads studied in this paper show that the accept strategy has a bigger impact on throughput under the one packet workload than with the SPECweb99-like workload. This is especially important in light of recent studies that have highlighted deficiencies of the SPECweb99 workload.

Nahum [18] analyzes the characteristics of the SPECweb99 workload in comparison with data gathered from several real-world web server logs. His analysis reveals many important shortcomings of the SPECweb99 benchmark. For example, the SPECweb99 benchmark does not use conditional GET requests, which account for 28% of all requests in some server traces, and often result in the server transmitting an HTTP header and no file data. Nahum also reports that SPECweb99's 5,120 byte median file size is significantly larger than the 230 bytes observed in one popular log. The combinations of these observations indicates that the demand for new connections at web servers is likely to be much higher than the demand generated by a SPECweb99-like workload.

Further evidence for this conclusion is provided by Jamjoom et al. [14]. They report that many popular browsers issue multiple requests for embedded objects in parallel. This is in contrast to using a single sequential persistent connection to request multiple objects from the same server. This strategy results in between 1.2 and 2.7 requests per connection which is considerably lower than the average of 7.2 requests per connection used by SPECweb99.

While a SPECweb99-like workload is still useful for measuring web server performance, it has a number of shortcomings and should not be used as the sole measure of server performance. Our one-packet workload highlights a number of phenomena (small transfer sizes, a small number of requests per connection) reported in recent literature. More importantly, as implemented by CNN.com, this is perhaps the best way to serve the most clients under conditions of extreme overload. In our work, it is useful because it places high demands on the server to accept new connections.

8 Conclusions

This paper examines the impact of connection-accepting strategies on web server performance. We devise and study a simple method for altering the accept strategy of three architecturally different servers: the user-mode single process event-driven μ server, the user-mode multi-threaded Knot server, and the kernel-mode TUX server.

Our experimental evaluation of different accept strategies expose these servers to representative workloads involving high connection-rates, and genuine overload conditions. We find that the manner in which each server accepts new connections can significantly affect its peak throughput and overload performance. Our experiments demonstrate that

well-tuned accept policies can yield noticeable improvements compared with the base approach used in each server. Under two different workloads, we are able to improve throughput by as much as 19% – 36% for TUX, 0% – 32% for Knot, and 39% – 71% for the μ server. As a result, we point out that researchers in the field of server performance must be aware of the importance of different accept strategies when comparing different types of servers.

Lastly, we present a direct comparison of the user-mode μ server and the kernel-mode TUX server. We show that the gap between user-mode and kernel-mode architectures may not be as large as previously reported. In particular, we find that under the workloads considered the throughput of the user-mode μ server rivals that of TUX.

In future work we plan to examine techniques for making more informed decisions about how to schedule the work that a server performs. We believe that by making more information available to the server we can implement both better and dynamic policies for deciding whether the server should enter a phase of accepting new connections (the accept-phase) or working on existing connections (the work-phase). Additionally this information would permit us to implement more controlled policies by limiting the length of each phase.

9 Acknowledgments

We gratefully acknowledge Hewlett Packard (through the Gelato Federation), the Ontario Research and Development Challenge Fund, and the National Sciences and Engineering Research Council of Canada for financial support for portions of this project. This work has benefited substantially from discussions with and feedback from Martin Arlitt, Michal Ostrowski, Brian Lynn, Amol Shukla, Ken Salem, Mohammed Abouzour and our shepherd Vivek Pai.

References

- [1] M. Arlitt and T. Jin. Workload characterization of the 1998 World Cup web site. *IEEE Network*, 14(3):30–37, May/June 2000.
- [2] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.
- [3] G. Banga and J.C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.
- [4] G. Banga, J.C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [5] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *Proceedings of the*

- USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey CA, December 1997.
- [6] T. Brecht and M. Ostrowski. Exploring the performance of select-based Internet servers. Technical Report HPL-2001-314, HP Labs, November 2001.
 - [7] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, 2001.
 - [8] Computer Science and Telecommunications Board. *The Internet Under Crisis Conditions: Learning from September 11*. The National Academies Press, 2003.
 - [9] Frank Dabek, Nickolai Zeldovich, M. Frans Kaashoek, David Mazieres, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 186–189, September 2002.
 - [10] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.
 - [11] HP Labs. The userver home page, 2003. Available at <http://hpl.hp.com/research/linux/userver>.
 - [12] E. Hu, P. Joubert, R. King, J. LaVoie, and J. Tracey. Adaptive fast path architecture. *IBM Journal of Research and Development*, April 2001.
 - [13] J. Hu, I. Pyarali, and D. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*. IEEE, November 1997.
 - [14] Hani Jamjoom and Kang G. Shin. Persistent dropping: An efficient control of traffic aggregates. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.
 - [15] Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John Tracey. High-performance memory-based Web servers: Kernel and user-space performance. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 175–188, 2001.
 - [16] C. Lever, M. Eriksen, and S. Molloy. An analysis of the TUX web server. Technical report, University of Michigan, CITI Technical Report 00-8, Nov. 2000.
 - [17] D. Mosberger and T. Jin. httpf: A tool for measuring web server performance. In *The First Workshop on Internet Server Performance*, pages 59–67, Madison, WI, June 1998.
 - [18] Eric Nahum. Deconstructing SPECWeb99. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, August 2002.
 - [19] M. Ostrowski. A mechanism for scalable event notification and delivery in Linux. Master's thesis, Department of Computer Science, University of Waterloo, November 2000.
 - [20] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.
 - [21] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.
 - [22] N. Provos and C. Lever. Scalable network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2000.
 - [23] N. Provos, C. Lever, and S. Tweedie. Analyzing the overload behavior of a simple web server. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, October 2000.
 - [24] Red Hat, Inc. *TUX 2.2 Reference Manual*, 2002.
 - [25] Standard Performance Evaluation Corporation. *SPECWeb99 Benchmark*, 1999. <http://www.specbench.org/osg/web99>.
 - [26] W.R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison Wesley, 1994.
 - [27] Thiemo Voigt and Per Gunningberg. Handling multiple bottlenecks in web servers using adaptive inbound controls. In *Proceedings of the International Workshop on Protocols For High-Speed Networks*, Berlin, Germany, April 2002.
 - [28] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea for high-concurrency servers. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.
 - [29] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.
 - [30] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.
 - [31] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, Banff, Oct. 2001.
 - [32] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T. Morris, David Mazieres, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX 2003 Annual Technical Conference*, June 2003.

Lazy Asynchronous I/O For Event-Driven Servers

Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox

Department of Computer Science

Rice University, Houston, Texas 77005, USA

{kdiaa,anupamc,alc}@cs.rice.edu

Willy Zwaenepoel

School of Computer and Communication Sciences

EPFL, Lausanne, Switzerland

willy.zwaenepoel@epfl.ch

Abstract

We introduce *Lazy Asynchronous I/O* (LAIO), a new asynchronous I/O interface that is well suited to event-driven programming. LAIO is *general* in the sense that it applies to all blocking I/O operations. Furthermore, it is *lazy* in the sense that it creates a continuation only when an operation actually blocks, and it notifies the application only when a blocked operation completes in its entirety. These features make programming high-performance, event-driven servers using LAIO considerably easier than with previous interfaces.

We describe a user-level implementation of LAIO, relying only on kernel support for scheduler activations, a facility present in many Unix-like systems.

We compare the performance of web servers implemented using LAIO to the performance obtained with previous interfaces. For workloads with an appreciable amount of disk I/O, LAIO performs substantially better than the alternatives, because it avoids blocking entirely. In one such case, the peak throughput with LAIO is 24% higher than the next best alternative. For in-memory workloads it performs comparably.

1 Introduction

We introduce *Lazy Asynchronous I/O* (LAIO), a new asynchronous I/O interface that is well suited to event-driven programs, in particular high-performance servers.

To achieve the best possible performance, an event-driven server must avoid blocking on any type of operation, from I/O to resource allocation. Thus, in an event-driven server, the use of asynchronous or non-blocking I/O is a practical

necessity. Asynchronous and non-blocking I/O support in present Unix-like systems is, however, limited in its generality. Non-blocking I/O can be performed on network connections, but not on disk files. POSIX asynchronous I/O (AIO) [11] can be performed on disk files, but only supports reading and writing. Many widely-used operations that require disk access as a part of their implementation, such as opening a file or determining its size, have no asynchronous equivalents.

In principle, this problem could be addressed by changes to the operating system. Such changes would affect the operating system's interface as well as its implementation. In practice, the scope of such changes has impeded such a solution. As a consequence, developers faced with this problem have either (1) abandoned an event-driven architecture entirely for a multithreaded or multiprocess architecture, (2) accepted that some operations can block and the effect thereof on performance, or (3) simulated asynchronous I/O at user-level by submitting blocking operations to a queue that is serviced by a pool of threads.

The tradeoff between multithreaded and event-driven servers has received considerable attention [5, 6, 7, 13]. Event-driven servers exhibit certain advantages including greater control over scheduling, lower overhead for maintaining state, and lower overhead for synchronization. Recently, von Behren et al. [12] have argued that compiler support and non-preemption can enable multithreaded servers to achieve performance comparable to event-driven servers, but such compiler support is not generally available.

Surprisingly, the second option does appear in practice. The `thttpd` web server [8] is a notable example, but as we show in Section 9, performance suffers as a result of blocking.

The *asymmetric multiprocess event-driven*

(AMPED) architecture that was employed by the Flash web server is representative of the third category [7]. In essence, it is a hybrid architecture that consists of an event-driven core augmented by helper processes. Flash performs all non-blocking operations in an event-driven fashion and all potentially blocking operations are dispatched to helper processes.

Unlike non-blocking I/O and AIO, LAIO is *general*. LAIO offers a non-blocking counterpart for each blocking system call, thereby avoiding the tradeoffs and the programming difficulties present with previous asynchronous I/O systems.

In addition to its generality, LAIO offers *lazy continuation creation*. If a potentially blocking system call completes without blocking, no continuation is created, avoiding the implementation cost of its creation and the programming complexity of dealing with it. Lazy continuation creation distinguishes LAIO from AIO in which executing an AIO primitive always creates a continuation, regardless of whether the call blocks or not. Furthermore, the LAIO API provides an event notification when a blocked call is *completed in its entirety*. This feature distinguishes LAIO from non-blocking I/O in which a return from a non-blocking call may indicate partial completion and in which the application may need to maintain state related to that partial completion.

Our implementation of LAIO resides entirely in a user-level library, without modification to the operating system's kernel. It requires support in the kernel for scheduler activations [3] to deliver upcalls to the LAIO library when an I/O operation blocks or unblocks.

The contributions of this paper are three-fold. First, we introduce a new asynchronous I/O interface, which is easier to program and performs better than previous I/O interfaces for workloads with an appreciable amount of disk I/O. Second, we document the ease of programming by comparing LAIO to non-blocking I/O, AIO and AMPED. For LAIO, non-blocking I/O, and AMPED, we quantify this comparison by counting the lines of affected code in the Flash web server [7]. Third, we evaluate the performance of LAIO by comparing the performance of two web servers, `httpd` [8] and Flash. We show that LAIO reduces blocking compared to non-blocking I/O and asynchronous I/O and incurs less overhead than AMPED.

The remainder of this paper is organized as follows. Section 2 describes the LAIO API. Section 3 provides an example of using LAIO. Sec-

tion 4 discusses other I/O APIs and their use in event-driven programs, and compares them with LAIO. Section 5 describes our LAIO implementation. Section 6 describes our experimental environment. Section 7 characterizes the performance of our implementation using a set of microbenchmarks. Section 8 describes the web server software and the workloads used in our macrobenchmarks. Section 9 describes the experimental results obtained with these macrobenchmarks. Section 10 discusses related work. Section 11 concludes this paper.

2 The LAIO API

The LAIO API consists of three functions: `laio_syscall()`, `laio_gethandle()`, and `laio_poll()`.

`laio_syscall()` has the same signature as `syscall()`, a standard function for performing indirect system calls. The first parameter identifies the desired system call. Symbolic names for this parameter, representing all system calls, are defined in a standard header file. The rest of the parameters correspond to the parameters of the system call being performed. If the system call completes without blocking, the behavior of `laio_syscall()` is indistinguishable from that of `syscall()`. If, however, the system call is unable to complete without blocking, `laio_syscall()` returns `-1`, setting the global variable `errno` to `EINPROGRESS`. Henceforth, we refer to this case as “a background `laio_syscall()`.” In such cases, any input parameters that are passed by reference, such as a buffer being written to a file, must not be modified by the caller until the background `laio_syscall()` completes.

`laio_gethandle()` returns an opaque handle for the purpose of identifying a background `laio_syscall()`. Specifically, this handle identifies the most recent `laio_syscall()` by the calling thread that reported `EINPROGRESS`. If the most recent `laio_syscall()` completed without blocking, `laio_gethandle()` returns `NULL`. `laio_gethandle()` is expected to appear shortly after the return of a `laio_syscall()` with return value `-1` and with `errno` equal to `EINPROGRESS`.

`laio_poll()` waits for the completion of background `laio_syscall()` operations. When one or more such operations have completed before a caller-specified timeout expires, `laio_poll()` returns a set of *LAIO com-*

pletion objects, one per completed background `laio_syscall()`. A completion object consists of a handle, a return value, and a possible error code. The handle identifies a background `laio_syscall()`, as returned by `laio_gethandle()`. The return value and possible error code are determined by the particular system call that was performed by the background `laio_syscall()`.

3 An LAIO Example

We present an example demonstrating how to write an event-driven program using LAIO. In this example, we also use *libevent* [9], a general-purpose event notification library. In general, such libraries provide a unifying abstraction for the various mechanisms that apply to different types of events. We have extended *libevent* to support completion of a background `laio_syscall()` as an event type. LAIO can, however, be used in isolation or with other event notification libraries.

Three functions from the *libevent* interface appear in this and subsequent examples: `event_add()`, `event_del()`, and `event_set()`. All of these functions work with an *event object* that has three principal attributes: the object being monitored, the desired state of that object, and a handler that is called when this desired state is achieved. For example, an event might designate a handler that is called when a socket has data available to be read. An event is initialized using `event_set()`. For the programmer's convenience, `event_set()` also accepts as its final argument an opaque pointer that is passed to the handler. `event_add()` enables monitoring for an initialized event. Unless an event is initialized as *persistent*, monitoring is disabled when the event occurs. `event_del()` disables monitoring for events that are either persistent or have not occurred.

In an event-driven program, event monitoring is performed by an infinite event loop. For each occurrence of an event, the event loop dispatches the corresponding event handler. Figure 1 shows the outline of the event loop in a program using LAIO. `laio_poll()` is used for event monitoring. It returns a set of LAIO completion objects, one for each completed background `laio_syscall()`. For each LAIO completion object, the event loop locates the corresponding event object (code not shown in the figure). It then invokes the continuation function

stored in that event object with the arguments returned in the LAIO completion object.

Figure 2 presents an event handler that writes data to a network socket. The write operation is initiated using `laio_syscall()`. If the write does not block, the number of bytes written is returned. The execution continues to `client_write_complete()`, and the event handler returns to the event loop.

In the more interesting case in which the write blocks, `-1` is returned and `errno` is set to `EINPROGRESS`. The program calls `laio_gethandle()` to get the handle associated with the background `laio_syscall()` operation. It initializes an event object, and associates the continuation function `client_write_complete()` with that LAIO handle. The event handler then returns to the event loop. The continuation function is invoked by the event loop after the background `laio_syscall()` completes (see Figure 1).

4 Comparison to Other I/O APIs

We describe non-blocking I/O and AIO and explain their usage in event-driven programming. We compare the LAIO API with these other I/O APIs.

4.1 Non-blocking I/O

With non-blocking I/O the programmer declares a socket as non-blocking, and then performs the normal I/O system calls on it. If the operation does not block, the operation returns normally. Otherwise, the operation returns when some part of the operation has completed. The amount of work completed is indicated by the return value.

Figures 3 and 4 illustrate how the event loop and the event handler presented in Figures 1 and 2 are programmed using non-blocking I/O instead of LAIO.

The event loop is conceptually the same as with LAIO. For non-blocking I/O, polling in the event loop can be done using any standard event monitoring mechanism like `select()`, `poll()`, or `kevent()`.

The event handler clearly illustrates the consequences of the fact that to avoid blocking the system call may return with the operation only partially completed. In this case the return value is different from the number of bytes provided as an argument to the system call. The application needs to remember how much of the data has

```

for (;;) {
    ***
    /* poll for completed LAIO operations; laioc_array is an array of LAIO completion
     * objects; it is an output parameter */
    if ((ncompleted = laio_poll(laioc_array, laioc_array_len, timeout)) == -1)
        /* handle error */
    for (i = 0; i < ncompleted; i++) {
        ret_val = laioc_array[i].laio_return_value;
        err_val = laioc_array[i].laio_errno;
        /* find the event object for laioc_array[i].laio_handle */
        eventp->ev_func(eventp->ev_arg/* == clientp */, ret_val, err_val);
        /* disable eventp; completions are one-time events */
    }
    ***
}

```

Figure 1: Event loop using LAIO

```

client_write(struct client *clientp)
{
    ***
    /* initiate the operation; returns immediately */
    ret_val = laio_syscall(SYS_write, clientp->socket, clientp->buffer,
        clientp->bytes_to_write);
    if (ret_val == -1) {
        if (errno == EINPROGRESS) {
            /* instruct event loop to call client_write_complete() upon completion
             * of this LAIO operation; clientp is passed to client_write_complete() */
            event_set(&clientp->event, laio_gethandle(), EV_LAIO_COMPLETED,
                client_write_complete, clientp);
            event_add(&clientp->event, NULL);
            return; /* to the event loop */
        } else {
            /* client_write_complete() handles errors */
            err_val = errno;
        }
    } else {
        err_val = 0;
        /* completed without blocking */
        client_write_complete(clientp, ret_val, err_val);
    }
    ***
}

```

Figure 2: Event handler using LAIO

been written. When the operation unblocks, an event is generated, and the event handler is called again, with the remaining number of bytes to be written. Several such write system calls may be required to complete the operation. Hence, the event is defined as persistent and deleted by the event handler only when the operation has completed.

4.2 AIO

The AIO API provides asynchronous counterparts, `aio_read()` and `aio_write()`, to the standard read and write system calls. In addition, `aio_suspend()` allows the program to wait for AIO events, and `aio_return()` and `aio_error()` provide the return and the error

values of asynchronously executed calls. An AIO control block is used to identify asynchronous operations.

Figures 5 and 6 show the outline of the event loop and an event handler, respectively, using AIO. The event loop is very similar to LAIO, except that two separate calls, to `aio_error()` and to `aio_return()`, are necessary to obtain the error and return values.

The more important differences are in the event handler in Figure 6. First, an AIO control block describing the operation is initialized. This control block includes information such as the descriptor on which the operation is performed, the location of the buffer, its size and the completion notification mechanism. Then, the operation is initiated. In the absence of errors, the event han-

```

for (;;) {
    ***
    /* poll for fds that are ready to read and/or write; pfd_array is an array of
     * pollfd objects listing blocked fds; it is an input and output parameter */
    if ((nready = poll(pfd_array, pfd_array_len, timeout)) == -1)
        /* handle error */
    for (i = 0; nready > 0 && i < pfd_array_len; i++) {
        if (pfd_array[i].revents & (POLLIN | POLLOUT)) {
            if (pfd_array[i].revents & POLLIN) { /* ready to read */
                /* find the read event object for pfd_array[i].fd */
                eventp->ev_func(eventp->ev_arg/* == clientp */);
            }
            if (pfd_array[i].revents & POLLOUT) { /* ready to write */
                /* find the write event object for pfd_array[i].fd */
                eventp->ev_func(eventp->ev_arg/* == clientp */);
            }
            nready--;
        }
    }
    ***
}

```

Figure 3: Event loop using non-blocking I/O

dler returns to the event loop afterwards.

The control block is used as a handle to identify the asynchronous operation. In most implementations, the application determines that the operation has finished through polling or an asynchronous event notification mechanism, such as signals. Figure 5 illustrates polling using the `aio_suspend()` operation.

4.3 Comparison

The three key distinguishing features of LAIO are:

Generality LAIO works for all operations, e.g., file and socket operations.

Laziness LAIO creates a continuation only if the operation blocks.

Completion notification LAIO notifies the application when the event completes, not at some intermediate stage.

A principal difference between LAIO on one hand and non-blocking I/O and AIO on the other hand is that LAIO allows any system call to be executed asynchronously. Non-blocking I/O only supports sockets, but does not support file operations. AIO only supports basic I/O operations, like reading and writing. Common operations that include I/O as part of their implementation, such as opening a file or determining its size, are not supported.

Furthermore, asynchronous I/O systems can be distinguished along two dimensions: whether or

not they create continuations if the operation does not block, and whether they provide a completion or a partial completion notification. LAIO creates continuations lazily and provides a completion notification. The examples in Sections 3, 4.1 and 4.2 clearly show the benefits of this combination in terms of programmability. Non-blocking I/O provides partial completion notification, requiring the application to maintain state about non-blocking calls in progress and to issue multiple I/O calls (see Figure 4). AIO creates continuations eagerly, requiring extra programming effort and extra system calls even if the operation does not block (see Figures 5 and 6).

5 An LAIO Implementation

LAIO is implemented as a user-level library.

The LAIO library maintains a pool of free LAIO handles. Each handle is an opaque pointer. The library passes an LAIO handle to the current running thread through an opaque pointer field in the thread structure. This is done via the KSE interface [10] before making any `laio_syscall()`. The library remembers the current handle in a variable `current_handle`. The library maintains another variable `background_handle` which is initialized to `NULL` before any `laio_syscall()`.

`laio_syscall()` is a wrapper around any system call. It saves the current thread's context and enables upcalls to be delivered. It then invokes the system call. If the operation does not block, it returns immediately to the appli-

```

client_write(struct client *clientp)
{
    ***
    /* assume that the one-time operations, enabling non-blocking I/O and
     * initializing the state of progress, have been performed elsewhere. */
    ***
    /* attempt the operation; returns immediately */
    ret_val = write(clientp->socket, &clientp->buffer[clientp->bytes_written],
        clientp->bytes_remaining);
    if (ret_val == clientp->bytes_remaining) { /* this write has completed */
        err_val = 0;
    } else if (ret_val > 0) { /* and implicitly less than bytes_remaining */
        if (clientp->bytes_written == 0) {
            /* instruct event loop to call client_write whenever clientp->socket
             * is ready to write; clientp is passed to client_write() */
            event_set(&clientp->event, clientp->socket, EV_PERSIST | EV_WRITE,
                client_write, clientp);
            event_add(&clientp->event, NULL);
        }
        /* update the state of progress */
        clientp->bytes_written += ret_val;
        clientp->bytes_remaining -= ret_val;
        return; /* to the event loop */
    } else if (ret_val == -1 && errno != EAGAIN) {
        /* client_write_complete() handles errors */
        err_val = errno;
    }
    if (clientp->bytes_written != 0) {
        /* instruct libevent that calls are no longer needed */
        event_del(&clientp->event);
    }
    client_write_complete(clientp, ret_val, err_val);
    ***
}

```

Figure 4: Event handler using non-blocking I/O

cation with the corresponding return value and upcalls are disabled. If the operation blocks, an upcall is generated by the kernel. The kernel creates a new thread and delivers the upcall on this new thread. At this point the thread associated with the handle `current_handle` has blocked. This handle is remembered in `background_handle` which now corresponds to the background `laio_syscall()`. Since, the current running thread has now changed, `current_handle` is set to a new handle from the pool of free LAIO handles. This new value of `current_handle` is associated with the current running thread via the KSE interface. `laio_gethandle()` returns the handle corresponding to the most recent background `laio_syscall()` which, as explained above, is saved in `background_handle`. The upcall handler then steals the blocked thread's stack using the context previously saved by `laio_syscall()`. Running on the blocked thread's stack, the upcall handler returns from `laio_syscall()` with the return value set to `-1` and the `errno` set to `EINPROGRESS`.

Unblocking of a background `laio_syscall()` generates another upcall. The upcall returns the handle corresponding to the unblocked thread via an opaque pointer field within the thread structure. The library adds this handle to a list of handles corresponding to background `laio_syscall()`s that have completed and frees the thread. The application calls `laio_poll()` to retrieve this list. After `laio_poll()` retrieves a handle it is returned to the pool of free LAIO handles.

We rely on scheduler activations [3] to provide upcalls for the implementation of the LAIO library. Many operating systems support scheduler activations, including FreeBSD [10], NetBSD [14], Solaris, and Tru64.

6 Experimental Environment

All of our machines have a 2.4GHz Intel Xeon processor, 2GB of memory, and a single 7200RPM ATA hard drive. They are connected by a gigabit Ethernet switch. They run FreeBSD

```

for (;;) {
    ***
    /* poll for completed AIO operations; aiocbp_array is an array of pointers
     * to the unfinished aiocbs; it is an input parameter */
    if (aio_suspend(aiocbp_array, aiocbp_array_len, timeout) == -1)
        /* handle error */
    for (i = 0; i < aiocbp_array_len; i++) {
        err_val = aio_error(aiocbp_array[i]);
        if (err_val == 0) { /* this aiocbp has completed */
            ret_val = aio_return(aiocbp_array[i]);
            /* find the event object for this aiocbp */
            eventp->ev_func(eventp->ev_arg/* == clientp */, ret_val, err_val);
            /* disable eventp; completions are one-time events */
        } else if (err_val == EINPROGRESS) { /* this aiocbp has not completed */
            continue;
        } else
            /* handle error */
    }
    ***
}

```

Figure 5: Event loop using AIO

5.2-CURRENT which supports *KSE*, FreeBSD's scheduler activations implementation.

7 Microbenchmarks

In order to compare the cost of performing I/O using LAIO, non-blocking I/O, and AIO, we implemented a set of microbenchmarks. These microbenchmarks measured the cost of 100,000 iterations of reading a single byte from a pipe under various conditions. For AIO, the microbenchmarks include calls to `aio_error()` and `aio_return()` in order to obtain the read's error and return values, respectively. We used a pipe so that irrelevant factors, such as disk access latency, did not affect our measurements. Furthermore, the low overhead of I/O through pipes would emphasize the differences between the three mechanisms. In one case, when the read occurs a byte is already present in the pipe, ready to be read. In the other case, the byte is not written into the pipe until the reader has performed either the `laio_syscall()` or the `aio_read()`. In this case, we did not measure the cost of a non-blocking read because the read would immediately return `EAGAIN`.

As would be expected, when the byte is already present in the pipe before the read, non-blocking I/O performed the best. LAIO was a factor of 1.4 slower than non-blocking I/O; and AIO was a factor of 4.48 and 3.2 slower than non-blocking I/O and LAIO, respectively. In the other case, when the byte was not present in the pipe before the read, we found that LAIO was a factor of 1.08

slower than AIO.

In these microbenchmarks, only a single byte was read at a time. Increasing the number of bytes read at a time, did not change the ordering among LAIO, non-blocking I/O, and AIO as to which performed best.

8 Macrobenchmarks

We use web servers as our benchmark applications. We make two sets of comparisons. First, we compare the performance of single-threaded event-driven servers using different I/O libraries, in particular using non-blocking I/O, AIO and LAIO. Second, we compare the performance of an event-driven server augmented with helper processes to the performance of a single-threaded event-driven server using LAIO.

We use two existing servers as the basis for our experiments, `thttpd` [8] and `Flash` [7]. We modify these servers in various ways to obtain the desired experimental results. We first describe `thttpd` and `Flash`. We then document the changes that we have made to these servers. We conclude with a description of the workloads used in the experiments.

8.1 `thttpd`

`thttpd` has a conventional single-threaded event-driven architecture. It uses non-blocking network I/O and blocks on disk I/O. All sockets are configured in non-blocking mode. An event is received in the event loop when a socket becomes

```

client_write(struct client *clientp)
{
    ***
    /* initialize the control block */
    aiocbp->aio_fildes = clientp->socket;
    aiocbp->aio_buf = clientp->buffer;
    aiocbp->aio_nbytes = clientp->bytes_to_write;
    aiocbp->aio_sigevent.sigev_notify = SIGEV_NONE; /* do nothing; event loop polls
                                                    * for completion */
    ***
    /* initiate the operation; returns immediately */
    if (aio_write(aiocbp) == -1) {
        /* client_write_complete() handles errors */
        client_write_complete(clientp, -1, errno);
    } else {
        /* instruct event loop to call client_write_complete() upon completion
         * of the AIO operation; clientp is passed to client_write_complete() */
        event_set(&clientp->event, aiocbp, EV_AIO_COMPLETED,
            client_write_complete, clientp);
        event_add(&clientp->event, NULL);
        return; /* to the event loop */
    }
    ***
}

```

Figure 6: Event handler using AIO

ready for reading or writing, and the corresponding event handler is invoked. From these event handlers `thttpd` makes calls to operations that may block on disk I/O, which may in turn cause the entire server to block.

We use the following terminology to refer to the different versions of the servers. A version is identified by a triple server-network-disk. For instance, **thttpd-NB-B** is the `thttpd` server using non-blocking network I/O and blocking disk I/O, as described in the previous paragraph.

8.2 Flash

Flash employs the asymmetric multiprocess event driven (AMPED) architecture. It uses non-blocking I/O for networking, and helper processes for operations that may block on disk I/O. All potentially blocking operations like file open or read are handled by the helper processes. The event-driven core dispatches work to the helper processes through a form of non-blocking RPC [2]: after the event-driven core sends a request to a helper process, it registers an event handler for execution upon receipt of the response. Using the notation introduced in Section 8.1, we refer to this server as **Flash-NB-AMPED**.

We use the most recent version of Flash, which uses `sendfile()`. The event-driven core reads the requested URL after accepting a connection. The corresponding file is opened by a helper process. Then, the event-driven core uses `send-`

`file()` to write the file to the corresponding socket. If `sendfile()` blocks in the kernel on a disk I/O, it returns a special `errno`. The event-driven core catches this `errno`, and instructs a helper process to issue explicit I/O to bring the file data into memory. After the event-driven core receives the helper process's response, indicating that the file data is in memory, it re-issues the `sendfile()`. Flash thus performs I/O in a lazy manner, analogous to LAIO. It calls `sendfile()` expecting it not to block, but if it blocks on disk I/O, the `sendfile()` call returns with a special error code, which is used to initiate I/O via helper processes.

8.3 Introducing LAIO

We modify `thttpd` to use the LAIO API, both for its network and disk operations. Blocking sockets are used for networking. All blocking system calls are invoked via `laio_syscall()`. Continuation functions are defined to handle `laio_syscall()` invocations that block and finish asynchronously. This version is called **thttpd-LAIO-LAIO**.

We first modify Flash to get a conventional single-threaded version. This version has no helper threads. Instead, all helper functions are called directly by the main thread. Non-blocking sockets are used as before to perform network I/O. We refer to this version as **Flash-NB-B**.

We then modify `Flash-NB-B` to use LAIO.

All sockets are configured as blocking, and potentially blocking operations are issued via `laio_syscall()`. This version is referred to as **Flash-LAIO-LAIO**.

8.4 Additional Versions for Comparison

We further modify Flash-NB-B to use AIO for file reads. This version of Flash uses sockets in a non-blocking mode. Since there is no AIO API for `stat()` and `open()`, it may block on those operations. We call this version of Flash **Flash-NB-AIO**.

We also modify Flash-NB-B to use LAIO for file I/O only. We call this version of Flash **Flash-NB-LAIO**.

Finally, we modify Flash-NB-AMPED to use kernel-level threads instead of processes to implement the helpers. Thus, the event-driven core and the helper threads share a single address space, reducing the cost of context switches between them. We call this version of Flash **Flash-NB-AMTED**.

8.5 Summary of Versions

The versions considered in the rest of the paper are summarized in Table 1.

8.6 Workloads

We use two trace-based web workloads for our evaluation. These workloads are obtained from the web servers at Rice University (Rice workload) and the University of California at Berkeley (Berkeley workload).

Table 2 shows the characteristics of the two workloads. The Rice and Berkeley workloads contain 245,820 and 3,184,540 requests, respectively. The columns *Small*, *Medium*, and *Large* indicate the percentage of the total number of bytes transferred in small files (size less than or equal to 8 Kilobytes), medium-sized files (size in between 8 Kilobytes and 256 Kilobytes), and large files (size greater than 256 Kilobytes). For the purposes of our experiments the most important characteristic is the working set of the workload: 1.1 Gigabytes for the Rice workload and 6.4 Gigabytes for the Berkeley workload. With the server machine we are using, which has 2 Gigabytes of main memory, the Rice workload fits in memory, while the Berkeley workload does not.

The trace file associated with each workload contains a set of request sequences. A sequence consists of one or more requests. Each sequence begins with a connection setup and ends with a connection teardown. Requests in a sequence are sent one at a time, the response is read completely, and then the next request is sent. In Flash, which supports persistent HTTP connections, all requests in a sequence are sent over a persistent connection. For `thttpd`, which does not support persistent connections, a new connection is set up before each request and torn down afterwards.

We use a program that simulates concurrent clients sending requests to the web server. The number of concurrent clients can be varied. Each simulated client plays the request sequences in the trace to the server. The program terminates when the trace is exhausted, and reports overall throughput and response time.

For each experiment we show a *cold cache* and a *warm cache* case. In the cold cache case, the cache is empty when the experiment is started. In the warm cache case, the cache is warmed up by running the entire experiment once before any measurements are collected.

9 Results

First, we compare single-threaded event-driven servers using different I/O libraries. Then, we compare LAIO to the Flash AMPED architecture with user-level helper processes.

9.1 Single-Threaded Event-Driven Servers

9.1.1 LAIO vs. Non-blocking I/O

Figure 7(a) shows the throughput for the Berkeley workload for `thttpd-NB-B`, `thttpd-LAIO-LAIO`, `Flash-NB-B` and `Flash-LAIO-LAIO` in both the cold and warm cache cases. `thttpd-LAIO-LAIO` achieves between 12% and 38% higher throughput than `thttpd-NB-B`, depending on the number of clients. `Flash-LAIO-LAIO` achieves between 5% and 108% higher throughput than `Flash-NB-B`. Figure 7(b) shows the response times for the four servers under the Berkeley workload. In both the cold and warm cache cases, `thttpd-LAIO-LAIO` and `Flash-LAIO-LAIO` achieve lower response times than `thttpd-NB-B` and `Flash-NB-B`, respectively. The Berkeley workload is too large to fit in memory. Thus, `thttpd-NB-B` and `Flash-NB-B` frequently

Server	Threaded	Blocking Operations	Comments
thttpd-NB-B	Single	disk I/O	stock version conventional event-driven
thttpd-LAIO-LAIO	Single		normal LAIO
Flash-NB-AMPED	Process-based Helpers		stock version multiple address spaces
Flash-NB-B	Single	disk I/O	conventional event-driven
Flash-LAIO-LAIO	Single		normal LAIO
Flash-NB-AIO	Single	disk I/O other than read/write	
Flash-NB-LAIO	Single		
Flash-NB-AMTED	Thread-based Helpers		single, shared address space

Table 1: Different versions of the servers

Web Workload	No. of requests	Small (≤ 8 KB)	Medium (> 8 KB and ≤ 256 KB)	Large (> 256 KB)	Total footprint
Rice	245,820	5.5%	20.2%	74.3%	1.1 Gigabytes
Berkeley	3,184,540	8.2%	33.2%	58.6%	6.4 Gigabytes

Table 2: Web trace characteristics

block on disk I/O regardless of whether the cache is cold or warm, while thttpd-LAIO-LAIO and Flash LAIO-LAIO do not.

Figure 8(a) shows the throughput for the Rice workload for the same servers in both the cold and warm cache cases. For the cold cache case, thttpd-LAIO-LAIO achieves between 9% and 36% higher throughput than thttpd-NB-B, depending on the number of clients. Flash-LAIO-LAIO achieves between 12% and 38% higher throughput than Flash-NB-B. No gain is achieved in the warm cache case. Figure 8(b) shows the response times. Similarly, thttpd-LAIO-LAIO and Flash-LAIO-LAIO only achieve lower response times than thttpd-NB-B and Flash-NB-B, respectively, in the cold cache case. For this workload, which fits in memory, LAIO shows gains in throughput and response time only in the cold cache case, stemming from compulsory misses, which block in the absence of LAIO. In the warm cache case, there is no disk I/O, and therefore no improvement as a result of using LAIO.

We conclude that LAIO substantially improves the performance of conventional single-threaded servers using non-blocking I/O. As explained in Section 4.1, LAIO is also easier to program than non-blocking I/O.

9.1.2 Additional Comparisons

One may wonder whether even better performance results could be obtained by using non-blocking I/O for the network and LAIO for file I/O. Figures 9(a) and 9(b) show the throughput results for Flash-LAIO-LAIO and Flash-NB-LAIO, for the Berkeley and the Rice workloads, respectively. The results for thttpd and for the response times follow the same general trend, so from now on we only show throughput results for Flash. The throughputs of Flash-NB-LAIO and Flash-LAIO-LAIO are very close in all cases. Given that no performance improvements result from using non-blocking I/O for networking, and given the simpler programming model offered by LAIO, we argue that it is better to use LAIO uniformly for all I/O.

In our final experiment we measure the difference between using LAIO uniformly for all I/O versus using non-blocking sockets for network I/O and AIO for disk operations. Figures 10(a) and 10(b) show the throughput of Flash-NB-AIO and Flash-LAIO-LAIO for the Berkeley and Rice workloads, respectively. For the Berkeley workload, Flash-LAIO-LAIO dominates Flash-NB-AIO after 128 clients, achieving its largest improvement of 34% at 512 clients. This is because

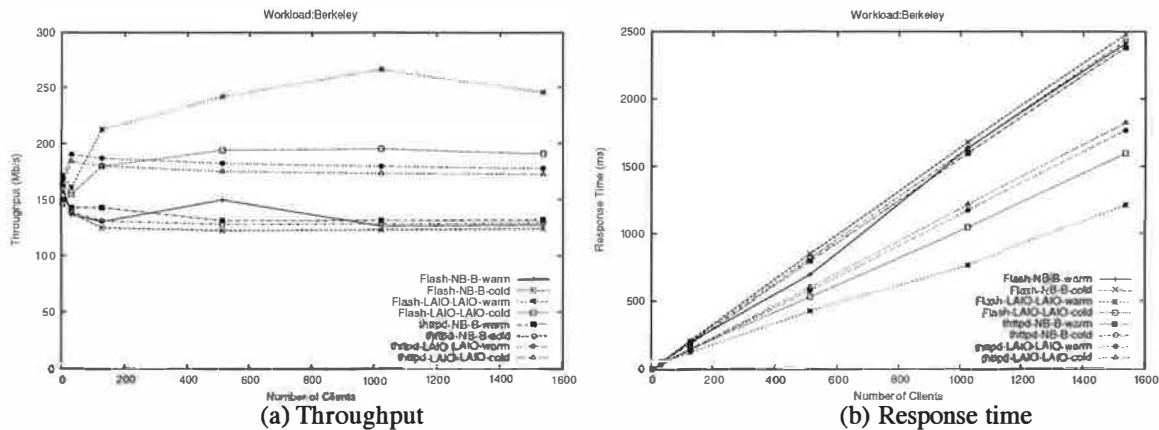


Figure 7: Results for the Berkeley workload with single-threaded event-driven web servers

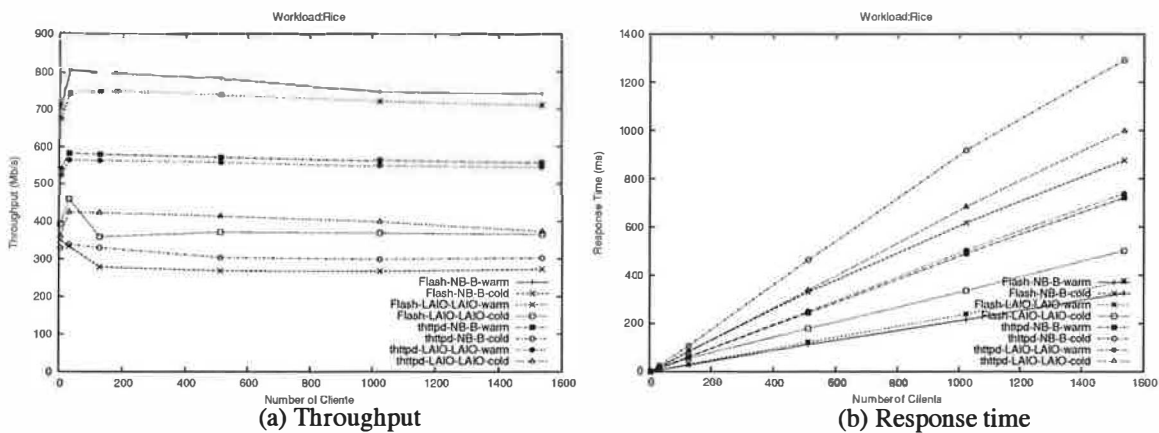


Figure 8: Results for the Rice workload with single-threaded event-driven web servers

AIO does not support asynchronous versions of `open()` and `stat()`. For the Rice workload, there is little difference between the two for either the cold or warm cache cases. In the cold cache case, the small number of files in the Rice workload results in relatively few of the operations that AIO does not support. In the warm cache case, the working set is in memory, so there is no disk I/O.

We conclude that the unified programming model offered by LAIO is preferable over mixed models combining non-blocking sockets and AIO or LAIO, both in terms of ease of programming and in terms of performance.

9.2 AMPED vs. Single-Threaded with LAIO

We compare the AMPED architecture with the event-driven architecture using LAIO for all I/O operations. First, we contrast these two architec-

tures from the view point of programming complexity. Next, we compare the performance of these two architectures. We use the Flash web server for these results.

9.2.1 Programming Complexity of AMPED vs. LAIO

We compare the programming complexity of the AMPED architecture to the LAIO API. We use lines of code as a quantitative measure of programming complexity.

By using LAIO we avoid the coding complexities of using helper processes and communicating between the main process and helper processes. We also avoid the state maintenance for incomplete write operations on sockets. As a result the programming effort is much reduced while using LAIO. This is illustrated in Table 3 which shows the lines of code associated with the different components of Flash for Flash-NB-AMPED

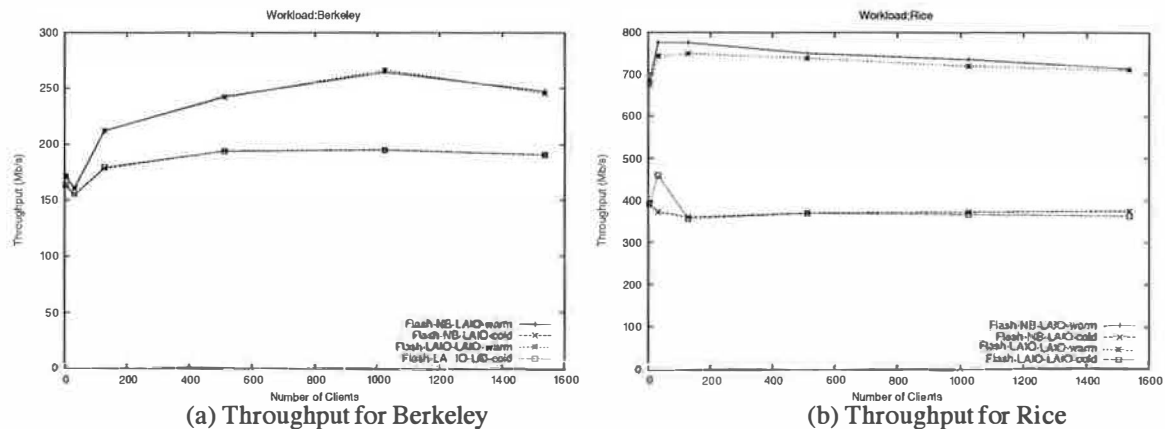


Figure 9: Results for Flash using non-blocking sockets and LAIO for disk operations

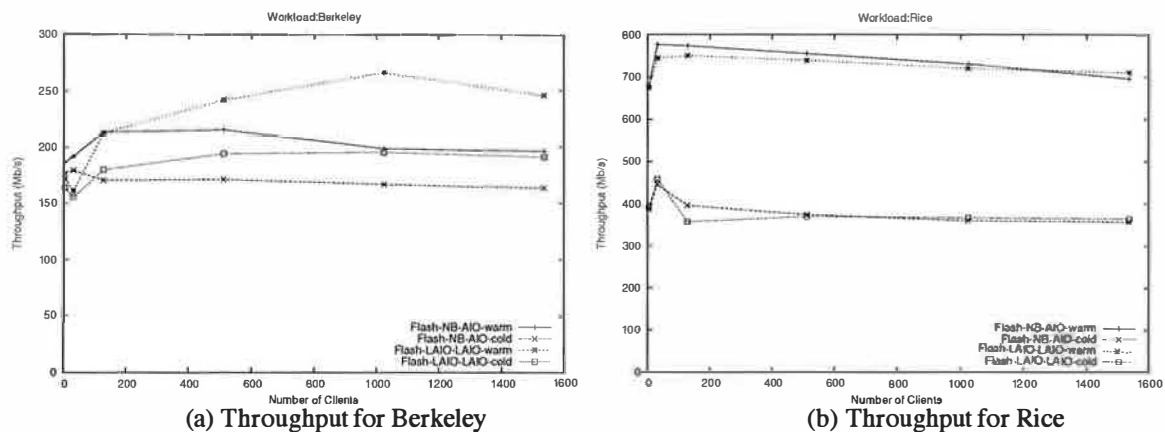


Figure 10: Results for Flash using non-blocking sockets and AIO for disk operations

and Flash-LAIO-LAIO.

Flash-NB-AMPED has two helper processes, the read helper and the name conversion helper. The read helper issues a `read()` on the file from disk, while the name conversion helper does `stat()` on the file and path name and checks permissions. The read helper accounts for 550 lines of code in Flash-NB-AMPED. In Flash-LAIO-LAIO the read helper code results in less than 100 lines of code. The name conversion helper required more than 600 lines of code in Flash-NB-AMPED. The name conversion function requires less than 400 lines of code in Flash-LAIO-LAIO. Eliminating the partial-write state maintenance results in a saving of 70 lines of code in Flash-LAIO-LAIO.

Considering the three components of Flash where Flash-NB-AMPED differs from Flash-LAIO-LAIO, Flash-NB-AMPED has more than three times the number of lines of code as Flash-LAIO-LAIO. Excluding comment lines

and blank lines, Flash-NB-AMPED has about 8,860 lines of code in total. Flash-LAIO-LAIO has about 8,020 lines of code. So in Flash-LAIO-LAIO we have reduced the code size by almost 9.5%.

9.2.2 Performance Comparison

Figure 11(a) shows the performance of Flash-NB-AMPED, Flash-NB-AMTED and Flash-LAIO-LAIO for the Berkeley workload under cold and warm cache conditions. Flash-LAIO-LAIO outperforms both Flash-NB-AMPED and Flash-NB-AMTED by about 10% to 40%. There is no noticeable performance difference between Flash-NB-AMPED and Flash-NB-AMTED. The Berkeley workload does not fit in memory and the CPU is not the bottleneck for any of these servers. The better performance of Flash-LAIO-LAIO comes from its better disk utilization than Flash-NB-AMPED or Flash-NB-AMTED. We

Component	Flash-NB-AMPED	Flash-LAIO-LAIO
File read	550	15
Name conversion	610	375
Partial-write state maintenance	70	0
Total code size	8860	8020

Table 3: Lines of code count for different versions of Flash

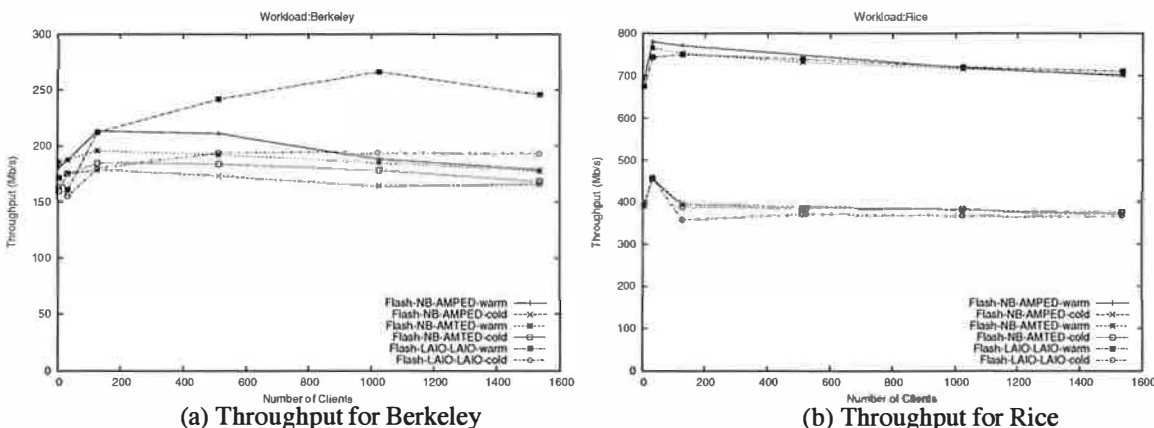


Figure 11: Results for Flash using the AMPED architecture

validate this by measuring the disk I/O statistics during a run of Flash-LAIO-LAIO and Flash-NB-AMPED under warm cache conditions. In particular, we measure total disk I/O (in bytes transferred) and average disk transfer rate (in bytes transferred per second). Flash-LAIO-LAIO transfers about 15.6 Gigabytes of data from the disk, which is about 15.7% lower than Flash-NB-AMPED which transfers about 18.5 Gigabytes. Flash-LAIO-LAIO achieves a disk transfer rate of 5.64 Mbytes/sec, which is about 5.8% higher than Flash-NB-AMPED which achieves a disk transfer rate of about 5.33 Mbytes/sec. Thus, Flash-LAIO-LAIO utilizes the disk more efficiently which translates into its higher performance. The disk I/O statistics for Flash-NB-AMTED are similar to that of Flash-NB-AMPED. Since the CPU was not the bottleneck, the context switching overhead (between address spaces) in Flash-NB-AMPED does not degrade its performance compared to Flash-NB-AMTED, and as such both these servers have almost the same performance.

Figure 11(b) shows the performance of Flash-NB-AMPED, Flash-NB-AMTED and Flash-LAIO-LAIO for the Rice workload under cold and warm cache conditions. Performance of these three servers is almost the same. The Rice

workload fits in memory. Hence, there is no disk I/O with a warm cache and the three servers perform the same. There is disk I/O starting from a cold cache, but the amount is much smaller than for the Berkeley workload. Flash-NB-AMPED and Flash-NB-AMTED perform almost the same because the CPU is not the bottleneck in the cold cache case and no I/O is required by the helpers in the warm cache case.

We conclude that an event-driven server using LAIO outperforms a server using the AMPED (or AMTED) architecture when the workload does not fit in memory, and matches the performance of the latter when the workload fits in memory.

10 Related Work

Some prior work has been done in this area. Other than AIO, the Windows NT [4] and VAX/VMS [1] operating systems have provided asynchronous I/O.

In Windows NT, the application can start an I/O operation then do other work while the device completes the operation. When the device finishes transferring the data, it interrupts the application's calling thread and copies the result to its address space. The kernel uses a Windows NT asynchronous notification mechanism called

asynchronous procedure call (APC) to notify the application's thread of the I/O operation's completion.

Like NT, VAX/VMS allows for a process to request that it gets interrupted when an event occurs, such as an I/O completion event. The interrupt mechanism used is called an asynchronous system trap (AST), which provides a transfer of control to a user-specified routine that handles the event.

Similar to AIO, asynchronous notifications in both VAX/VMS and Windows NT are limited to a few events, mainly I/O operations and timers. This is not broad enough to support any system call like in LAIO. Also asynchronous I/O in either Windows NT or VAX/VMS is not lazy.

11 Conclusions

We have introduced Lazy Asynchronous I/O (LAIO), a new asynchronous I/O interface that is well suited to event-driven programming, particularly the implementation of high-performance servers. LAIO is general in that it supports all system calls, and lazy in the sense that it only creates a continuation if the operation actually blocks. In addition, it provides notification of completion rather than partial completion.

LAIO overcomes the limitations of previous I/O mechanisms, both in terms of ease of programming and performance. We have demonstrated this claim by comparing LAIO to non-blocking I/O, AIO and AMPED.

By means of an example we have shown the programming advantages of LAIO over the alternatives. Furthermore, we have quantified the comparison between LAIO, non-blocking I/O, and AMPED by counting the affected lines of code within the Flash web server. Flash-LAIO-LAIO, a version of Flash using LAIO for both networking and disk, has 9.5% fewer lines of code than Flash-NB-AMPED, the stock version of Flash using a combination of non-blocking I/O for networking and AMPED for disk.

We have experimented with two web servers, `thttpd` and Flash, to quantify the performance advantages of LAIO. We have shown that for workloads which cause disk activity LAIO outperforms all alternatives, because it avoids blocking in all circumstances and because it has low overhead in the absence of blocking. For one such workload, Flash-LAIO-LAIO achieved a peak throughput that was 25% higher than Flash-NB-AMPED and 24% higher than the next best

event-driven version, Flash-NB-AIO, using non-blocking I/O for networking and AIO for disk. Under workloads with no disk activity, there was little difference in throughput among the servers.

References

- [1] *VAX Software Handbook*. Digital Equipment Corporation, 1981.
- [2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In *Proceedings of the USENIX Annual Technical Conference (NO 98)*, pages 143–154, June 1998.
- [3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the UserLevel Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.
- [4] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.
- [5] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *2002 USENIX Annual Technical Conference*, pages 103–114, June 2002.
- [6] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation at the 1996 USENIX Annual Technical Conference, Jan. 1996.
- [7] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.
- [8] J. Poskanzer. `thttpd - tiny/turbo/throttling http server`. Version 2.24 is available from the author's web site, <http://www.acme.com/software/thttpd/>, Oct. 2003.
- [9] N. Provos. Libevent - An Event Notification Library. Version 0.7c is available from the author's web site, <http://www.monkey.org/~provos/libevent/>, Oct. 2003. Libevent is also included in recent releases of the NetBSD and OpenBSD operating systems.
- [10] The FreeBSD Project. FreeBSD KSE Project. At <http://www.freebsd.org/kse/>.
- [11] The Open Group Base Specifications Issue 6 (IEEE Std 1003.1, 2003 Edition). Available from <http://www.opengroup.org/onlinepubs/007904975/>, 2003.
- [12] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003.
- [13] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, Oct. 2001.
- [14] N. Williams. An Implementation of Scheduler Activations on the NetBSD Operating System. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 99–108, June 2002.

Energy Efficient Prefetching and Caching*

Athanasios E. Papathanasiou and Michael L. Scott

University of Rochester

{papathan,scott}@cs.rochester.edu

<http://www.cs.rochester.edu/~papathan,~scott>

Abstract

Traditional disk management strategies—prefetching and caching in particular—are designed to maximize performance. In mobile systems they conflict with strategies that attempt to save energy by powering down the disk when it is idle. We present new rules for prefetching and caching that maximize power-down opportunities (without performance loss) by creating an access pattern characterized by intense bursts of activity separated by long idle times. We also describe an automatic system that monitors past application behavior in order to generate appropriate prefetching hints, and a general system of kernel enhancements that coordinate I/O activity across all running applications.

We have implemented our system in the Linux kernel, and have measured its performance and energy consumption via physical instrumentation of a running laptop. We describe our implementation and present quantitative results. For workloads including a mix of sequential access to large files (multimedia), concurrent access to large numbers of files (compilation), and random access to large files (speech recognition), we report disk energy savings of 60–80%, with negligible loss in throughput or interactive responsiveness.

1 Introduction

Prefetching and caching are standard practice in modern file systems. They serve to improve performance—to increase throughput and decrease latency—by eliminating as many I/O requests as possible, and by spreading the requests that remain as smoothly as possible over time. This strategy results in relatively short intervals of inactivity. It ignores the goal of energy efficiency so important to mobile systems, and in fact can frustrate that goal. Magnetic disks, network interfaces, and similar devices provide low-power states that save energy only when idle intervals are relatively long. A smooth access pattern can eliminate opportunities to save energy even during such light workloads as MPEG and MP3 playback.

The aim of our work is to create bursty access patterns for devices with non-operational low-power states, increasing the average length of idle intervals and maximizing utilization when the device is active, without compromising performance. At present we are focusing on hard disks.

Typical hard disks for mobile systems support at least four power states: Active, Idle, Standby, and Sleep. The disk only works in the Active state. In the Idle state the disk is still spinning, but the electronics may be partially unpowered, and the heads may be parked or unloaded. In the Standby state the disk is spun down. The Sleep state powers off all remaining electronics; a hard reset is required to return to higher states. Individual devices may support additional states. The IBM TravelStar, for example, has three different Idle sub-states.

One to three seconds are typically required to transition from Standby to Active state. During that spin-up time the disk consumes 1.5–2X as much power as it does when Active. The typical laptop disk must therefore remain in Standby state for a significant amount of time—on the order of 5–16 seconds for current laptop disks—to justify the energy cost of the subsequent spin-up. The energy savings in Idle state approaches that of Standby state, particularly in very small form factor devices, and the time and energy to move from Idle to Active state are minimal. Hence, even modest increases of the disk's idle interval can lead to significant energy savings.

In previous work we made the case for energy efficiency through burstiness and demonstrated the energy efficiency potential of aggressive prefetching for rate-based applications with sequential access patterns [28, 29]. In this paper, we provide a detailed description of the design and implementation of our prefetching and caching algorithms. In addition, we provide experimental results for more challenging workload mixes, including applications that make non-sequential accesses to multiple files.

We have implemented our system in the Linux 2.4.20 kernel. We have extended the memory management and file system of the operating system with algorithms and data structures to:

- Quickly identify the working set of the executing job

*This work was supported in part by NSF grants EIA-0080124, CCR-9988361, and CCR-0204344; by DARPA/AFRL contract F29601-00-K-0182; and by Sun Microsystems Laboratories.

mix and dynamically control the amount of memory used for aggressive prefetching and buffering of dirty data.

- Coordinate the generation of I/O requests among concurrently running applications, so that they are serviced by the device during the same small window of time.

In designing and evaluating our system we have focused on long-running applications that are commonly executed on mobile systems, and that generate a large number of I/O requests, separated by idle times too short to be exploited by the hard disk for energy savings. Examples include data transfer (copying); encoding, decoding, and compression; compilation and build; scripting utilities for system maintenance; and computationally demanding user interface tasks, e.g. speech recognition.

The following Section provides rules for optimal prefetching when the goal is energy efficiency. Sections 3 and 4 describe the design of our prefetching and request deferring mechanisms. Section 5 presents experimental results. Section 6 discusses previous work. Section 7 summarizes our conclusions.

2 Prefetching for Energy Efficiency

To illustrate the differences that arise when prefetching has the additional goal of improving disk energy efficiency, consider an application with reference string $\{A B C D E F G \dots\}$ and a steady rate of one access every 10 time units. Assume that the buffer cache has room for three blocks, and that the disk requires one time unit to fetch a block that misses in the cache.¹

Figures 1–3 illustrate the execution of the application while accessing the first 6 elements of the reference string, with an optimal replacement strategy and three different prefetching strategies: Figure 1 illustrates a *fetch-on-demand* strategy; Figure 2 illustrates a strategy that follows the prefetching rules of Cao et al. [1]; Figure 3 illustrates an energy-conscious prefetching strategy that attempts to maximize the length of the disk idle intervals. Under the *fetch-on-demand* strategy the application runs for 66 time units and experiences 6 misses. The disk idle time is spread across 6 intervals of 10 time units each. With a traditional prefetching strategy (Figure 2) run time decreases to 61 time units and the application experiences just one miss. The distribution of disk idle time remains practically unchanged, however: 5 intervals of 9 time units each and one of 8 time units. The energy-conscious prefetching strategy (Figure 3) achieves the same execution time and the same number of misses as traditional

¹Note that the application in this example consumes data at a rate slower than the bandwidth of the disk. The goal of our work is to increase the length of disk idle interval for workloads that run for relatively long periods and do not require the disk to be active constantly.

Time	Application	Disk	Cache State			
1		fetch(A)	<table><tr><td></td><td></td><td></td></tr></table>			
2–11	access(A)	idle	<table><tr><td>A</td><td></td><td></td></tr></table>	A		
A						
12		fetch(B)	<table><tr><td>A</td><td></td><td></td></tr></table>	A		
A						
13–22	access(B)	idle	<table><tr><td>A</td><td>B</td><td></td></tr></table>	A	B	
A	B					
23		fetch(C)	<table><tr><td>A</td><td>B</td><td></td></tr></table>	A	B	
A	B					
24–33	access(C)	idle	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C				
34		fetch(D)	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C				
35–44	access(D)	idle	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
45		fetch(E)	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
46–55	access(E)	idle	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
56		fetch(F)	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
57–66	access(F)	idle	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
67		fetch(G)	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				

Figure 1: Optimal Replacement and Fetch-on-Demand. Accessing the reference string up to element F requires 66 time units. The disk idle time appears in 6 intervals of 10 time units each.

prefetching, but the disk idle time appears in two much longer intervals: one of 27 time units and one of 28 time units.

2.1 Rules for Optimal Prefetching Revisited

Traditional prefetching strategies aim to minimize execution time by deciding:

- when to fetch a block from disk,
- which block to fetch, and
- which block to replace.

Previous work [1] describes four rules to make these decisions in a performance-optimal fashion:

1. *Optimal Prefetching*: Every prefetch should bring into the cache the next block in the reference stream that is not yet in the cache.
2. *Optimal Replacement*: Every prefetch should discard the block whose next reference is farthest in the future.
3. *Do no harm*: Never discard block A to prefetch block B when A will be referenced before B.
4. *First Opportunity*: Never perform a prefetch-and-replace operation when the same operations (fetching the same block and replacing the same block) could have been performed previously.

The first three rules answer the questions of what to prefetch (rules 1 and 2) and (partially) when to prefetch (rule 3), and apply equally well to energy-conscious prefetching. The fourth rule suggests that a prefetch operation should be issued when (a) a prior fetch completes,

Time	Application	Disk	Cache State			
1		fetch(A)	<table><tr><td></td><td></td><td></td></tr></table>			
2	access(A)	prefetch(B)	<table><tr><td>A</td><td></td><td></td></tr></table>	A		
A						
3	access(A)	prefetch(C)	<table><tr><td>A</td><td>B</td><td></td></tr></table>	A	B	
A	B					
4–11	access(A)	idle	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C				
12	access(B)	prefetch(D)	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C				
13–21	access(B)	idle	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
22	access(C)	prefetch(E)	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
23–31	access(C)	idle	<table><tr><td>D</td><td>E</td><td>C</td></tr></table>	D	E	C
D	E	C				
32	access(D)	prefetch(F)	<table><tr><td>D</td><td>E</td><td>C</td></tr></table>	D	E	C
D	E	C				
33–41	access(D)	idle	<table><tr><td>D</td><td>E</td><td>F</td></tr></table>	D	E	F
D	E	F				
42	access(E)	prefetch(G)	<table><tr><td>D</td><td>E</td><td>F</td></tr></table>	D	E	F
D	E	F				
43–51	access(E)	idle	<table><tr><td>G</td><td>E</td><td>F</td></tr></table>	G	E	F
G	E	F				
52	access(F)	prefetch(H)	<table><tr><td>G</td><td>E</td><td>F</td></tr></table>	G	E	F
G	E	F				
53–61	access(F)	idle	<table><tr><td>G</td><td>H</td><td>F</td></tr></table>	G	H	F
G	H	F				
62	access(G)	prefetch(I)	<table><tr><td>G</td><td>H</td><td>F</td></tr></table>	G	H	F
G	H	F				

Figure 2: Optimal Replacement and Traditional Prefetching. Accessing the reference string up to element F requires 61 time units. The disk idle time appears in 5 intervals of 9 time units each and one of 8. In a long run the average idle interval length will be 9 time units.



Time	Application	Disk	Cache State			
1		fetch(A)	<table><tr><td></td><td></td><td></td></tr></table>			
2	access(A)	prefetch(B)	<table><tr><td>A</td><td></td><td></td></tr></table>	A		
A						
3	access(A)	prefetch(C)	<table><tr><td>A</td><td>B</td><td></td></tr></table>	A	B	
A	B					
4–11	access(A)	 idle	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B		C			
12–21	access(B)		<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C				
22–30	access(C)	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C	
A	B	C				
31	access(C)	prefetch(D)	<table><tr><td>A</td><td>B</td><td>C</td></tr></table>	A	B	C
A	B	C				
32	access(D)	prefetch(E)	<table><tr><td>D</td><td>B</td><td>C</td></tr></table>	D	B	C
D	B	C				
33	access(D)	prefetch(F)	<table><tr><td>D</td><td>E</td><td>C</td></tr></table>	D	E	C
D	E	C				
33–41	access(D)	 idle	<table><tr><td>D</td><td>E</td><td>F</td></tr></table>	D	E	F
D	E		F			
42–51	access(E)		<table><tr><td>D</td><td>E</td><td>F</td></tr></table>	D	E	F
D	E	F				
52–60	access(F)	<table><tr><td>D</td><td>E</td><td>F</td></tr></table>	D	E	F	
D	E	F				
61	access(F)	prefetch(G)	<table><tr><td>D</td><td>E</td><td>F</td></tr></table>	D	E	F
D	E	F				

Figure 3: Optimal Replacement and Energy-conscious Prefetching. Accessing the reference string up to element F requires 61 time units. The disk idle time appears in one interval of 27 time units and one of 28. In a long run the average idle interval length will be 28 time units.

or (b) the block that would be discarded was just referenced. Identifying this first opportunity can be difficult, and indeed most real systems are considerably less aggressive. As noted by Patterson et al. [31], the full performance benefit of prefetching will be achieved even if the prefetch completes just barely before the corresponding access.

We observe, however, that any uniform prefetch policy will tend to produce a smooth access pattern, with short idle times. As an example, consider a system with a cache size of k blocks, a reference string $\mathcal{R} = \{b_1 b_2 \dots b_k b_{k+1} \dots b_n\}$, where $n > k$, and an inter-access time of \mathcal{A} . If we follow rule 4, we will fetch block b_{k+1} immediately after the reference to b_1 , b_{k+2} immediately after the reference to b_2 and so on, breaking a possible disk idle interval of length $k \times \mathcal{A}$ into intervals of length $\mathcal{A} - \mathcal{F}$, where \mathcal{F} represents the time to fetch a block from the disk. Assuming $\mathcal{F} < \mathcal{A}$, an energy-conscious prefetching algorithm should not initiate the prefetch of b_{k+1} until \mathcal{F} time units prior to its reference. Then in a single burst it should prefetch blocks $\{b_{k+1} \dots b_{2k-1}\}$, replacing blocks $\{b_1 \dots b_{k-1}\}$. Intuitively, this policy alternates between “first opportunity” and “just in time” prefetching, depending on whether the disk is currently in the Active state.

Based on the above discussion, to accommodate the requirement of energy efficiency, we replace rule 4 with the following:

- 4'. *Maximize Disk Utilization*: Always initiate a prefetch operation after the completion of a fetch, if there are blocks available for replacement (with respect to Rule 3).
- 5'. *Respect Idle Time*: Never interrupt a period of inactivity with a prefetch operation unless the prefetch has to be performed immediately in order to maintain optimal performance.

Rule 4' guarantees that a soon-to-be-idle disk will not be allowed to become inactive if there are blocks in the cache that may be replaced by blocks that will be accessed earlier in the future. This way disk utilization is maximized and short intervals of idle time that cannot be exploited for energy efficiency are avoided. Rule 5' attempts to maximize the length of a period of inactivity without degrading performance. Note that the rule implies that the prefetching algorithm should take into account additional delays due to disk activation or congestion as well as the time required for a fetch to complete. An algorithm that follows rules 4' and 5' will lead to the same hit ratio and execution time as an algorithm following the rules of Cao et al., but will exhibit fewer and longer periods of disk inactivity whenever possible.

State	Value
Active Power	2.0 W
Idle Power	0.61 W
Idle-to-Active Energy	1.5 J
Idle-to-Active Time	0.55 s
Active-to-Idle Energy	2.4 J
Active-to-Idle Time	0.85 s
Standby Power	0.15 W
Spin up Energy	5.0 J
Spin up Time	1.6 s
Spin down Energy	2.94 J
Spin down Time	2.3 s

Table 1: Abstract disk model parameters for computing the potential of aggressive speculative prefetching. Values are based on the characteristics of the Hitachi DK23DA hard disk.

2.2 Prefetching's Potential for Energy Savings

In comparison to traditional prefetching, which aims to reduce the latency of access to disks in the active state, prefetching for energy efficiency has to be significantly more aggressive in both quantity and coverage. A traditional prefetching algorithm can fetch data incrementally: its goal is simply to request each block far enough in advance that it will already be available when the application needs it. It will improve performance whenever its rate of “true positives” (prefetched blocks that turn out to indeed be needed) is reasonably high, and its “false positives” (prefetched blocks that aren’t needed after all) don’t get in the way of fetching the “false negatives” (needed blocks that aren’t prefetched). By contrast, an energy-reducing prefetching algorithm must fetch enough blocks to satisfy all read requests during a lengthy idle interval. Minimizing “false negatives” is more important than it is in traditional prefetching, since the energy cost and performance penalty of power state transitions is very high. These differences suggest the need to fetch much more data, and much more speculatively, than has traditionally been the case. Indeed, prefetching for burstiness more closely resembles prefetching for disconnected operation in remote file systems [19] than it does prefetching for low latency.

Fortunately, avoiding a disk power-up operation through speculative prefetching can justify fetching a large number of “false positives” in terms of energy consumption. To show the energy saving potential of speculative prefetching, we present calculated values for an aggressive, speculative prefetching algorithm that follows the rules for energy-efficient prefetching (Section 2.1) with various “false-positive” to “true-positive” ratios, and compare to a prefetching algorithm that follows Rules 1–4 and prefetches only what is needed. We assume an ab-

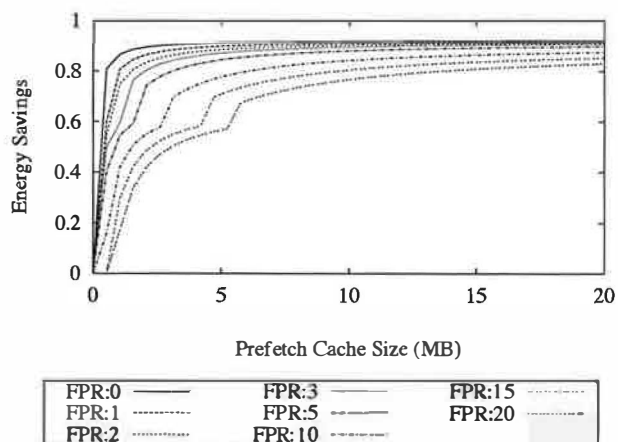


Figure 4: Energy savings of a speculative prefetching algorithm across various prefetch buffer sizes and “false-positive” to “true-positive” ratios (*FPR*) ranging from 0 to 20. An application with a “slow” data consumption rate of 16 KB/sec is assumed.

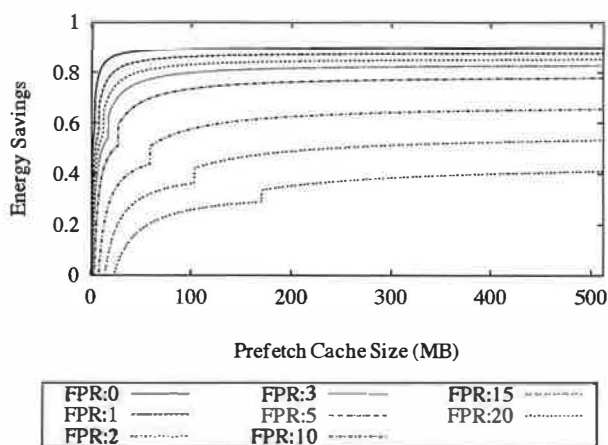


Figure 5: Energy savings of a speculative prefetching algorithm across various prefetch buffer sizes and “false-positive” to “true-positive” ratios (*FPR*) ranging from 0 to 20. An application with a “fast” data consumption rate of 240 KB/sec is assumed.

stract disk model based on the Hitachi DK23DA hard disk with the characteristics shown in Table 1 and an optimal power management policy (one that spins the disk down whenever it can save energy by doing so, without performance loss). We include in our calculations the cost of reading data into memory. Based on the specifications of Micron’s 512 Mb \times 16 SDRAM DDR dies [25], this is 100 μ J per page.

Figures 4 and 5 present results for “false-positive” to “true-positive” ratios (*FPR*) ranging from 0 to 20 for a pair of applications consuming data at 16 KB/s and 240 KB/s,

respectively. For the slower application (Figure 4), with even a small amount of memory dedicated to prefetching, significant energy savings can be achieved. Just 5 MB of memory prefetching leads to over 50% energy savings, even for “false-positive” to “true-positive” ratios as high as 20 to 1, i.e. even if we prefetch more than 20 times as much data as we actually use. For the faster application (Figure 5), a 50% savings in disk + memory energy can be achieved with a 25 MB prefetch buffer for “false-positive” to “true-positive” ratios of up to 5 to 1. Larger ratios require significantly more prefetch memory.

3 Design of Energy-Aware Prefetching

As mentioned in the previous Section, any prefetching algorithm has to decide *when to fetch a block, which block to fetch, and which block to replace*.

3.1 Deciding When to Prefetch

Based on Rules 4' and 5' presented in the previous Section, our prefetching algorithm attempts to fetch from the disk as many blocks that are going to be accessed in the future as possible during periods when the disk is active, and to postpone prefetching operations until the latest opportunity during periods when the disk is idle. To approximate such behavior, we introduce an Epoch-based algorithm into the memory management mechanisms of the operating system. Each epoch consists of two phases: an *active* phase and an *idle* phase. Prefetching occurs during the active phase. To manage this prefetching, the OS must:

1. Predict future data accesses. Prediction is based on manual or automatic hints (Section 3.2).
2. Compute the amount of memory that can be used for prefetching or storing new data. This step requires identifying quickly the currently useful in-memory data: the workload's working set and cached files.
3. Free the required amount of memory by unmapping pages and flushing dirty, mapped pages.
4. Prefetch or reserve buffers for writing new data proportional to each executing application's memory resource requirements. The goal of this step is to coordinate I/O accesses across concurrently running applications so that they all generate their next demand miss at approximately the same time.

When the active phase completes, the idle phase of the epoch begins. During the idle phase, accesses to each active file are monitored by the operating system. Based on the access pattern and rate, and the state of the buffer cache, the operating system attempts to predict the next miss time for each file, and to initiate a new prefetching cycle early enough to achieve optimal performance.

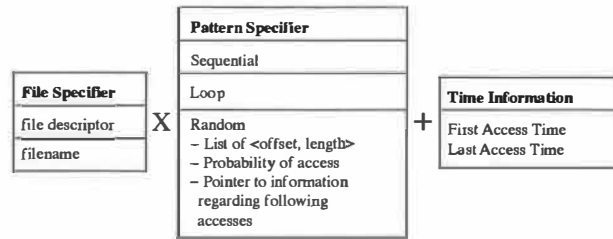


Figure 6: Hint Interface: Applications can disclose future file accesses using the hint interface. The interface specifies a file using a filename or a file descriptor and provides a pattern specifier that can be one of sequential, loop and random. In addition, an estimation of the time of first and last access can be given, if available.

Prefetching has to start well in advance in order to hide both the latency of the fetch itself and the delay of a possible disk reactivation (power-up).

The start of a new epoch is triggered by one of the following events:

1. A new prefetching cycle has to be initiated in order to maintain optimal performance.
2. A demand miss took place. In this case the prefetching algorithm has failed to load into memory all required data, or has mistakenly evicted useful pages from the buffer cache during the active phase. The application that issued the request may experience an increased delay in addition to the penalty of a demand miss if the disk has been placed into a low-power state.
3. The system is low on memory resources. The page freeing logic has to be executed.

3.2 Deciding What to Prefetch

To achieve as high a degree of accuracy as possible, prediction is based on *hints*. Our hint interface is an extension of that described by Patterson et al. [31]. The hint interface consists of a file specifier that can be a file descriptor or a filename, a pattern specifier that shows whether the file will be accessed in a sequential, loop or random way, and estimates of the times of the first and last accesses to the file. The time information can be represented as offsets from the start of the application execution or from the disclosure of the hint. For randomly accessed files the hint interface also provides a list of “hot” clusters within the file, the probability of access to each cluster, and an optional pointer to a file that provides sets of clusters that have a significant probability of being accessed within a certain time period of the access to a specific page. Such information can be generated through profiling. Figure 6 summarizes the hint interface. Hints are submitted to the operating system through a new set of system calls.

New applications can use the new system calls directly in order to provide hints to the operating system.

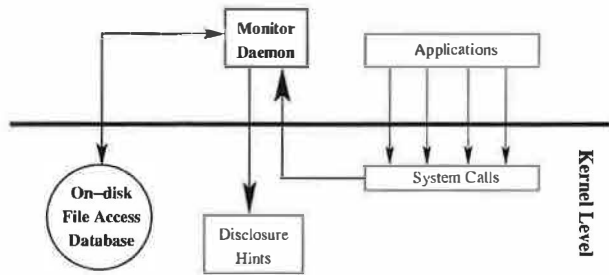


Figure 7: The monitor daemon provides hints automatically on behalf of applications that do not support the hint interface. It tracks file system use by monitoring system calls. It analyzes the collected information, and creates a hint database for each application whose access pattern may be harmful to energy efficiency. When an application with hints re-executes, its file accesses will automatically be disclosed to the operating system by the monitor daemon.

For existing applications, efficient file prediction may be achieved by monitoring past file accesses and taking advantage of the *semantic locality* that appears in user behavior [20]. In our current prototype, a *monitor daemon* tracks the file activity of all executing applications by tracing the open, close, read, write `execve`, `exit`, and `setpgid` system calls. Its goals are two-fold: to prepare a database describing file accesses for each application based on the collected information (*Access Analysis*) and to generate hints automatically on behalf of applications based on the information in the database (*Hint Generation*). Figure 7 illustrates the operation of the monitor daemon.

Since access analysis can be a computationally intensive operation, it takes place only at periods during which energy consumption is not a concern (when the mobile system is plugged in). The analysis utility discovers associations among file accesses and applications, and records the access pattern (sequential, loop, and random) and the time from the beginning of execution to the first and last access to each file. For randomly accessed files the utility also identifies clusters of pages within the file that tend to be accessed together, the probability of an access to a given cluster, and for each cluster X a list of clusters that tend to be accessed within a certain time (currently one minute) of the access to that cluster. The analysis utility stores this information for applications that use the file system for relatively long periods of time (currently at least 1 minute) and cause access patterns that spread a large amount of idle time across many short intervals. Occasional accesses that appear during interactive workloads can be handled adequately by previously proposed disk power management policies [4, 5, 15, 22].

In order to associate related file accesses that are gen-

erated by different processes (as an example consider accesses caused by the various invocations of the `gcc` compiler during the execution of a `make` operation), we take advantage of the process group structure of Unix. The analysis utility associates accesses with the process group leader of the process that caused the access. The database that maintains the hinting information is indexed by the absolute pathname of the executable with which the access was associated, the directory in which the access took place, and the arguments used to make the invocation.

During normal system operation, the monitor daemon tracks `execve` system calls. If there are hints available in the database for the specified application, the daemon automatically generates the hints on behalf of that application.

3.3 Deciding What to Replace

As mentioned in Section 3.2, the first step during the initiation of a new prefetching cycle is to compute the number of pages that can be used for prefetching. Unlike a traditional prefetching algorithm, which can make incremental page-out decisions over time, the energy-conscious prefetching algorithm must make a decision significantly ahead of time, predicting the number of cached pages that are not going to be accessed during a possibly long idle phase.

Intuitively, two parameters determine the number of pages that should be freed at the beginning of each epoch. First, the reserved amount of memory should be large enough to contain all predicted data accesses. Second, prefetching or future writes should not cause the eviction of pages that are going to be accessed sooner than the prefetched data. Since our goal is to maximize the length of the hard disk's idle periods, we use the type of the first miss during an epoch's idle phase to refine our estimate of the number of pages available for prefetching in the subsequent epoch.

We categorize page misses as follows:

1. **Eviction miss:** A miss on a page that used to reside in the buffer cache, but was evicted in favor of prefetching. Such a miss suggests that number of pages used for prefetching in the current epoch was too large.
2. **Prefetch miss:** A miss on a page for which there was a prediction (hint) that it was going to be accessed. Such a miss suggests that a larger prefetch cache size could have been used during the current epoch.
3. **Compulsory miss:** A miss on a page for which there is no prior information.

An eviction miss during the idle phase of an epoch suggests that the prefetching depth (the total number of pages prefetched) should decrease, while a prefetch miss suggests that the prefetching depth should increase. Controlling the prefetching depth based on eviction misses pro-

vides a way to protect the system from applications that may issue incorrect hints. Incorrect hints will increase the number of eviction misses and lead to a reduced prefetching depth. Section 4.4 describes in detail how our system adjusts the prefetching depth.

4 Implementation

We have implemented our epoch-based energy-efficient prefetching algorithm in the Linux kernel, version 2.4.20. We describe that implementation in this Section.

4.1 Hinted Files

Prefetching hints are disclosed by the monitor daemon or by applications themselves. In addition, the kernel automatically detects and generates hints for long sequential file accesses. The full set of hints, across all files is maintained in a doubly linked list, sorted by estimated first access time. In addition to the information listed in Section 3.2), the kernel keeps track of the following:

- *Memory Status*: Shows the caching state for a file. A file can be completely uncached, have only its metadata cached, be partially cached, or be fully cached. When a hint is disclosed using the filename of the file, the caching state of the corresponding file is unknown. In such cases the kernel assumes that the file is completely uncached, and aggressively attempts to prefetch metadata and data. The kernel also keeps track of open and close requests in order to associate hinted accesses with their corresponding file descriptors.
- *Rate of access*: Keeps track of the average rate in pages per second with which the file is being accessed.
- *Prefetch Depth*: Shows the number of pages that were used for prefetching data for the corresponding hinted file during the current epoch. The prefetching algorithm allocates memory for prefetching to each hinted file proportional to its average access rate.

4.2 Prefetch Thread

Idle interval length can be limited because of a lack of coordination among requests generated by different applications. Even if there are long idle periods in the access pattern of every application, we will be unable to power down the disk unless these patterns are in phase. The operating system must ensure that read and write requests from independent concurrently running applications are issued during the same small window of time. Write activity can easily be clustered because most write requests are issued by a single entity: the update daemon. Similarly, page-out requests are issued by the swap daemon. Read and prefetching requests, however, are generated within a process context independent of other applications. To coordinate prefetching requests across all running applications we introduce a centralized entity that is responsible for generating prefetching requests for all running ap-

plications: the prefetch daemon. The prefetch daemon is analogous to the update daemon and handles read activity. Through the prefetch thread the problem of coordinating I/O activity is reduced to that of coordinating three daemons.

During the active phase the prefetch thread goes through the list of hints and attempts to prefetch data in a way that will equalize the expected time to first miss across all applications. The prefetching algorithm sets an initial target idle period and for each hinted file that is predicted to be accessed within the target period it aggressively prefetches metadata and data proportional to the average access rate. The target idle time is then increased gradually until all hinted files are fully prefetched or the amount of memory available for prefetching is depleted.

The target idle times used by the prefetching algorithm are based on the “breakeven” times of modern mobile hard disks: the times over which the energy savings of low-power states exactly equal the energy cost of returning to the Active state. Currently, we are using target idle times of 2 seconds, which corresponds to the highest low-power state of the IBM TravelStar [17], 5 seconds, which corresponds to the IDLE-3 low-power state of the TravelStar and the Idle state of the Hitachi DK23DA, and multiples of the Standby state breakeven time (16 seconds for the DK23DA). When the prefetching cycle completes, the algorithm predicts that the length of the idle phase of the upcoming epoch will be the period for which the prefetch thread successfully prefetched all necessary data.

4.3 Prefetch Cache

We have augmented the kernel’s page cache with a new data structure: the prefetch cache. Pages requested by the prefetch daemon are placed in the prefetch cache. Each page in the prefetch cache has a timestamp that indicates when it is expected to be accessed. When a page is referenced, or its timestamp is exceeded, it is moved to the standard LRU list and is thereafter controlled by the kernel’s page reclamation policy.

4.4 Eviction Cache

To choose an appropriate size for the prefetch cache, we must keep track of pages that are evicted in favor of prefetching (Section 3.3). We do this using a new data structure called the *eviction cache*. This cache retains the metadata of recently evicted pages (though not their contents!) along with a unique serial number, called the *eviction number*. The eviction number counts the number of pages that have been evicted in favor of prefetching. During the idle phase, if an eviction miss takes place, the difference between the page’s eviction number and the current epoch’s starting eviction number indicates the number of pages that were evicted in favor of prefetching *without* causing an eviction miss. It can be used as an estimate

of a suitable prefetching depth (prefetch cache size) for the upcoming epoch. The prefetch depth does not change in the case of compulsory misses or misses on pages that were evicted in prior epochs. It is increased by a constant amount if there were no misses, or if there were only prefetch misses.² In order to avoid significant oscillations in the prefetching depth we use a moving average function. During system initialization, when there is no prior information available, the prefetch depth is set to the number of idle pages in the system.

4.5 Handling Write Activity

The original Linux kernel uses a variant of the approximate interval periodic update policy [26]. The update daemon runs every 5 seconds, and flushes all dirty buffers that are older than 30 seconds. This policy is bad for energy efficiency: under even light write workloads idle time will appear in intervals of 5 seconds or less.

In our current implementation, we use a modified update daemon that flushes all dirty buffers once per minute. In addition, we have extended the open system call with an additional flag that indicates that write-behind of dirty buffers belonging to the file can be postponed until the file is closed or the process that opened the file exits. Such a direction is useful for several common applications, such as compilations and MP3 encoding, that produce files that do not have strict intra-application reliability constraints. For legacy code, users can specify in a configuration file the names of applications (gcc, for example) or file classes (e.g. .o) for which write-back can be delayed. The monitor daemon then provides a “flush-on-close” or “flush-on-exit” directive to the operating system.

A side effect of create and write accesses is that they can lead to unpredicted read operations of file metadata, directories, or file system metadata (e.g. the free block bitmaps). Such read operations are synchronous and can interrupt lengthy idle intervals. For this reason the monitor daemon keeps track of write and create accesses and generates hints for any files that may be written or created by a certain application. During the prefetching cycle of the epoch, the prefetch thread speculatively prefetches file system metadata for files associated with write or create hints. At the memory management level of the operating system, file system structure is not known, since it is file system dependent. In order to enable file system metadata prefetching we have extended the Linux virtual file system with two new function pointers: `emulate_create` and `emulate_write`. Both functions execute the low level file system code that is normally executed during file creation or disk block al-

location without actually modifying the file system (only the corresponding read requests are issued). Currently, we have an implementation of the two functions for the Linux Ext2 file system.

Finally, write activity can lead to unexpected I/O requests during an idle phase if the system runs out of memory and the page daemon has to start paging. For this reason, during the active phase the prefetch thread reserves a portion of the available memory for future writes. The amount of memory allocated to each file is proportional to its write rate. At the end of the prefetching cycle, the prefetch threads clears a number of pages equal to the total number of reserved pages. Pages are reserved only for files that have been active for at least a certain time period (5 seconds) and have a fast write rate (at least 1 page per second).

4.6 Power Management Policy

At the completion of the prefetching cycle, the prefetch thread predicts the length of the upcoming idle phase (Section 4.2). This prediction is forwarded to the kernel’s power management policy. If the predicted length is longer than the hard disk’s Standby breakeven time, the disk is set to the Standby state within one second after it becomes idle (the disk may be servicing requests for several seconds after the prefetch thread completes its execution).

Since the prediction provided by the prefetching system is based only on file accesses associated with hints, there is a significant chance of decreased prediction accuracy during highly interactive workloads that are not handled efficiently by the monitor daemon. To avoid harmful spin-down operations, the power management algorithm monitors the accuracy of the prefetching system’s predictions. If the prefetching system repeatedly mispredicts the length of the idle phase, providing predictions that specify idle periods longer than the disk’s Standby breakeven time, when the actual idle period length is shorter than the breakeven time, the power management policy reverts to a dynamic-threshold spin-down policy, ignoring predictions coming from the prefetch thread until their accuracy is increased.

For rate-based and non-interactive applications, the same information that allows the operating system to identify opportunities for spin-down can also be used to predict appropriate times for spin-up, rendering the device available just in time to service requests. For this purpose during the idle phase of an epoch the operating system monitors the rate at which pages belonging to sequentially accessed files are consumed from the prefetch cache. Based on this rate, the number of pages remaining in the prefetch cache, and the disk’s power-up delay, it computes the time at which the disk has to be activated in order to avoid any noticeable delays.

²In the current implementation we increase by the `pages_low` value, used by the pageout daemon. In Linux this is 1/128 of the total number of memory pages but not less than 20 or more than 255.

Disk	Hitachi
Capacity	10-30GB
Active	2.1W
Active Idle	1.6W (24%)
Low-Power Idle	0.6W (71%)
Standby	0.15W (93%)
Spin up	3.0W
Spin-up time	1.6s
Breakeven time	16s

Table 2: Energy consumption parameters for the Hitachi DK23DA hard disk. The disk supports three low power states: Active Idle (a portion of the electronics is off), Low Power Idle (the heads are parked) and Standby.

5 Experimental Evaluation

In this section, we compare our energy-conscious prefetching algorithm (called *Bursty*) to the standard Linux 2.4.20 policies across systems with memory sizes ranging from 64 MB to 492 MB. We use the power management policy described in Section 4.6 for the Bursty system, and a 10 second fixed threshold power management policy for Linux. Linux 2.4.20 supports a conservative prefetching algorithm that reads up to 128 KB (32 4KB pages) in advance and leads to very short periods of disk idle time for our experimental workloads. Hence, the power management policy degenerates to the No-Spin-Down policy.

Our experiments were conducted on a Dell Inspiron 4100 laptop with 512 MB of total memory and a Hitachi DK23DA hard disk. Table 2 presents the power consumption specifications of the disk. Power measurements were collected from the disk's two 5V supply lines. To measure power, both the voltage and current need to be known. The voltage is assumed to be fixed at 5V. We used 100m Ω precision resistors in order to dynamically measure the current through the supply lines. The voltage drop across the resistors was measured through the following National Instruments setup: a SCXI-1308 voltmeter terminal block, a SCXI-1102C (32 channel multiplexer/amplifier and 10kHz filter) module, a SCXI-1000 chassis (for the mentioned modules), a SCXI-1349 card to chassis cable, and a PCI-6052E Analog-to-Digital converter card (capable of 16 bit resolution, 333Ksamples/second). The gain of the SCXI-1102C was set to 100 and the PCI-6052E was set to have a range of ± 10 V. The sampling rate for each signal was 1000 samples/second. Measurements were collected and converted to current and power using National Instrument's LabView (version 6.0.2) software.

The idle interval histogram graphs (Figures 8 and 9) are based on traces collected from the ATA/IDE disk driver during the execution of our workloads scenarios. In order to avoid any disk activity caused by the tracing system,

we used a pinned-down 20 MB memory buffer that was periodically transmitted to a logging system through the network.

We use four different workload scenarios, with different degrees of I/O intensity. The first, MPEG playback of two 76 MB files (referred to as *MPEG*), represents a relatively intensive read workload. The second (referred to as *Concurrent*) is a read and write intensive workload, which involves concurrent MP3 encoding and MPEG playback. The MP3 encoder reads 10 WAV files with a total size of 626 MB and produces 42.9 MB of data. During the MP3 encoding process, the MPEG player accesses two files with a total size of 152 MB. Our third workload (referred to as *Make*) is a full Linux kernel compilation. Finally, in order to evaluate our system for workloads that consist of random accesses to files, we use the SPHINX-II Speech Recognition utility [16] from CMU (referred to as *SPHINX*). During speech recognition SPHINX accesses a 128 MB language model file in an apparently random way. As input we use a set of recorded dialogues that were used in the evaluation of the TRIPS Interactive Planning System [8]. We used a subset of the dialogues to prepare the access pattern database (Section 3.2), and evaluated the system on a different subset.

The metrics used in the comparisons are:

Length of idle periods Longer idle periods can be exploited by more power-efficient device states. Increasing the length of idle periods can improve any underlying power management policy.

Energy savings We compare the energy savings achieved for Linux and our Bursty system for various memory sizes.

Slowdown A significant challenge for our Bursty system is to minimize the performance penalties that may be caused by increased disk congestion and disk spin-up operations.

Figures 8-11 show the distribution of idle time intervals for our workload scenarios. We present results for our Bursty system using various memory sizes. For the first three workloads, the memory size ranges from 64 MB to 492 MB. For SPHINX two sizes are used: 256 MB and 492 MB. Executing SPHINX on systems with less than 256 MB of memory leads to thrashing. In all graphs the straight vertical line represents the 16 second break-even point of the Hitachi hard disk. When executing on the standard Linux kernel (not shown in the graphs), the first three workloads lead to access patterns in which 100% of the disk idle time appears in intervals of less than 1 second, independent of memory size, preventing the use of any low-power mode. In contrast, larger memory sizes lead to longer idle interval lengths for the Bursty system, providing more opportunities for the disk to transition to a low-power mode. During the Linux kernel compilation,

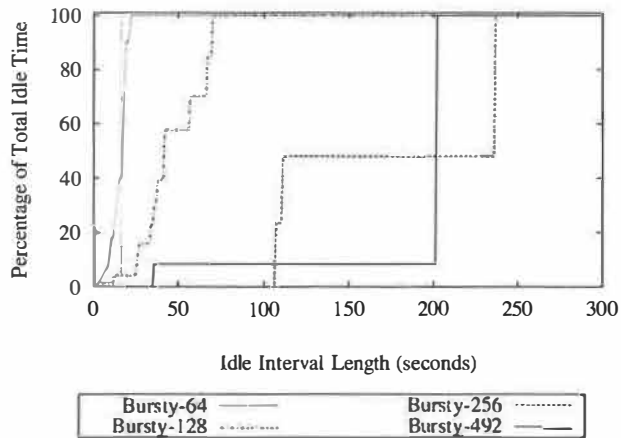


Figure 8: Cumulative distribution of disk idle time intervals during MPEG playback. On the standard Linux kernel (not shown), 100% of the disk idle time appears in intervals of less than 1 second for all memory sizes.

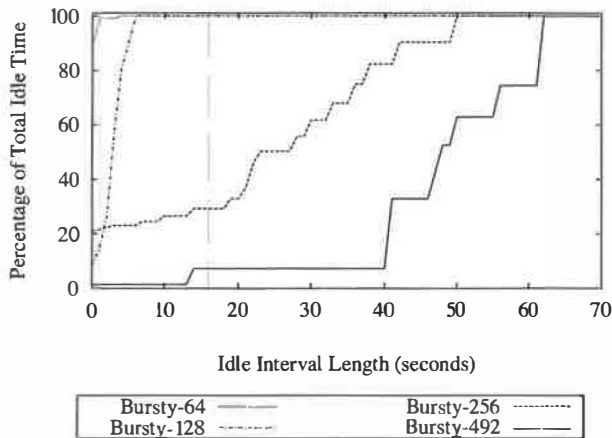


Figure 9: Distribution of disk idle time intervals during concurrent MPEG playback and MP3 encoding. On the standard Linux kernel (not shown), 100% of the disk idle time appears in intervals of less than 1 second for all memory sizes.

the Bursty system manages to prefetch most of the accessed data when system memory exceeds 128 MB. At 96 MB our energy-aware prefetching algorithm slowly increases the size of the prefetch cache, eventually achieving idle periods that are longer than the disk's breakeven time. The algorithm behaves similarly on a 64 MB system. However, it also leads to increased paging that has a negative effect on both energy and performance. For the speech recognition workload at 492 MB the Bursty system prefetched the whole language model file leading to long idle phases. At 256 MB it prefetched up to 33% of the file, leading to idle interval lengths only slightly longer

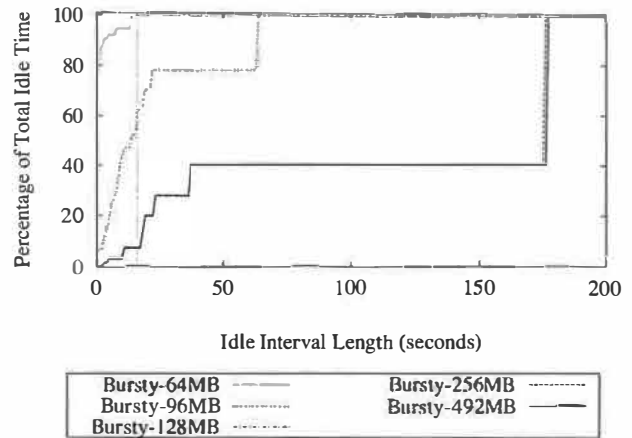


Figure 10: Distribution of disk idle time intervals during a full Linux kernel compilation. On the standard Linux kernel (not shown), 100% of the disk idle time appears in intervals of less than 1 second for all memory sizes. At a memory size of 128 MB and over all accessed files are prefetched by the Bursty system leading to increased idle interval lengths.

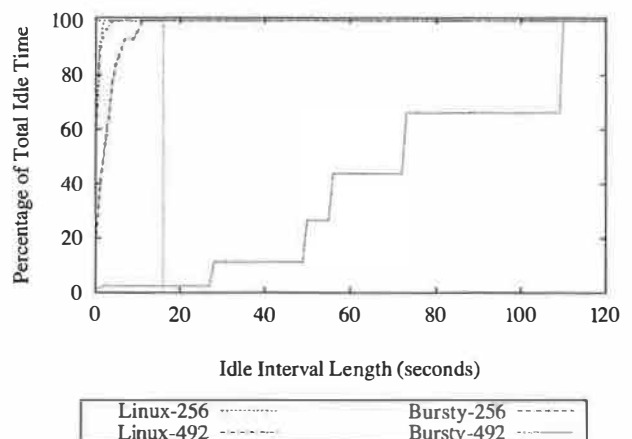


Figure 11: Distribution of disk idle time intervals during speech recognition by SPHINX. On the standard Linux kernel with 256 MB, 100% of the disk idle time appears in intervals of less than 1 second.

than Linux, due to accesses to the uncached portion of the file.

Figure 12 presents disk energy savings as a function of total system memory size. The base case used for the comparisons is the standard Linux kernel on a 64 MB system. For Linux, increasing the system's memory size has only a minor impact on the energy consumed by the disk, because of the lack of long idle intervals. In contrast, the savings achieved by the Bursty algorithm depend on the amount of memory available. For the first workload, sig-

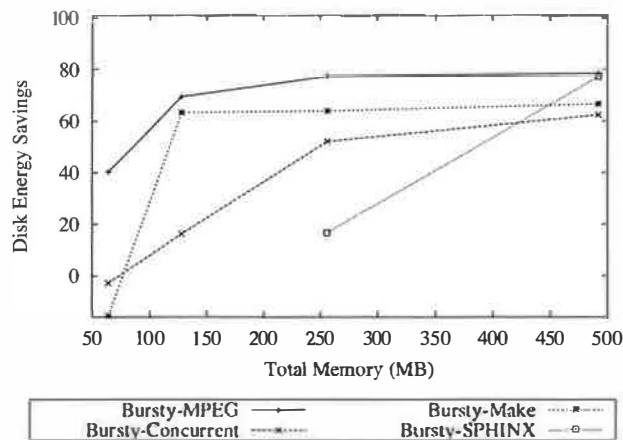


Figure 12: Disk energy savings as a function of total memory size. Results are shown for all experimental workloads: *MPEG* playback, *Concurrent* *MPEG* playback and *MP3* encoding, *Make* (the Linux kernel compilation) and *SPHINX* executing on our Bursty system. When executing on the standard Linux kernel (not shown), increasing the total memory size to 492 MB leads to at most 5.25% in disk energy savings across all workloads.

nificant energy savings are achieved for all memory sizes. Even on the 64 MB system, the energy consumed by the disk is reduced by 40.3%. Despite the fact that most disk idle intervals are not long enough to justify a spin-down operation, they allow the disk to make efficient use of the low-power idle state that consumes just 0.6 W. With 492 MB, the Bursty system loads the required data in just three very intensive I/O bursts, allowing the disk to transition and remain in the spin-down state for significant periods of time, and leading to 78.5% disk energy savings.

Results for the second workload are similar. However, because of the increased I/O intensity the energy savings are less pronounced. Energy consumption is reduced after the memory size exceeds 128 MB (15.9% energy savings). On a system with 492 MB energy savings reach 62.5%. For the Linux kernel compilation, our Bursty system achieved significant energy savings for memory sizes of 128 MB and larger: up to 66.6%. However, despite the increase in idle interval lengths, on a 64 MB system our algorithm leads to increased energy consumption (15.5%) because of excessive paging. Finally, for the speech recognition workload, disk energy savings reach 77.4% for a 492 MB system. With 256 MB, the energy-conscious prefetching algorithm saves 16.7% through more efficient use of the active idle power mode.

Figure 13 presents the execution time for the workload scenarios. For the *Concurrent* workload (left side of the graph), the slowdown of the *MP3* encoding process is 2.8% or less. The performance of the *MPEG* player

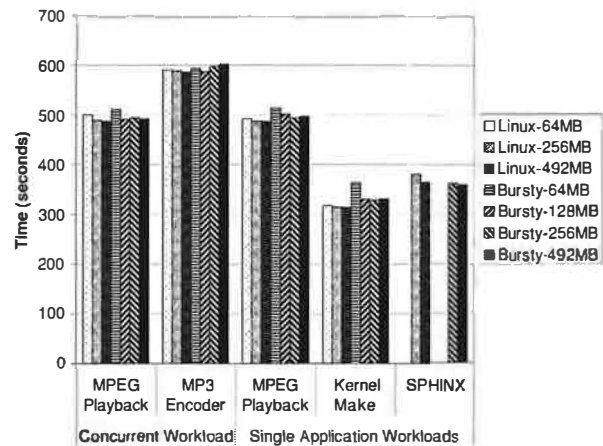


Figure 13: Execution time (in seconds) of the workloads on Linux and the Bursty system with various memory configurations.

stays within 1.6% of that on the Linux system in all cases except the 64 MB system (Bursty-64MB), where it experiences a slowdown of 4.8%. For the *MPEG* playback workload, the Bursty system experiences a slowdown of 1% (for the 64 MB case) or less, when compared to Linux with 492 MB of memory (Linux-492MB). For the Linux kernel compilation the Bursty system stays within 5% of the execution time of Linux across all memory sizes larger than 128 MB. On a 64 MB system Bursty experiences a performance penalty of 15% mostly due to increased paging and disk congestion. Using a priority-based disk queue that gives higher priority to demand requests than prefetching requests could lead to improved performance. Finally, during speech recognition aggressive prefetching of the language model file leads to slightly improved performance for *SPHINX* due to the reduction in page cache misses. Our performance results show that the prefetching algorithm manages to avoid successfully most of the delay that may be caused by disk spin-up operations. In addition, it can lead to improved performance because of an increased cache hit ratio.

6 Related Work

Power Management. The research community has been very active in the area of power-conscious systems during the last few years. Golding et al. [11] hint at the idea of conserving such non-renewable resources as battery power during idle periods by disabling unused resources. Ellis et al. [6] suggest making energy efficiency a primary metric in the design of operating systems. ECOSystem [37] provides a model for accounting and for fairly allocating the available energy among competing applications according to user preferences. Odyssey [10, 27] provides operating system support for application-aware

resource management. The key idea is to trade quality for resource availability.

Several policies have been proposed to decrease the power consumption of processors that support dynamic voltage and frequency scaling. The key idea is to schedule so as to “squeeze out the idle time” in rate-based applications. Several researchers have proposed voltage schedulers for general purpose systems [9, 12, 35, 32]. Lebeck et al. [21] explore power-aware page allocation in order to make more efficient use of memory chips supporting multiple power states, such as the Rambus DRAM chips.

Hard Disks. The energy efficiency of hard disks is not a new topic. The cost and risks of Standby mode played a role in the early investigation of hard-disk spin-down policies [4, 5, 15, 22]. Concurrently with our own work [28], several groups have begun to investigate the deliberate generation of bursty access patterns. Heath et al. [14] and Weissel et al. [36] propose user-level mechanisms to increase the burstiness of I/O requests from individual applications. Lu et al. [24] report that significant energy can be saved by respecting the relationship between processes and devices in the CPU scheduling algorithm. (We note, however, that given the many-second “break-even” times for hard disks, process scheduling can increase burstiness only for non-interactive applications, which can tolerate very long quanta.) Zeng et al. [38] propose “shaping” the disk access pattern as part of a larger effort to make energy a first-class resource in the eyes of the operating system.

Like Lu et al. and Zeng et al., we believe that the effective management of devices with standby modes requires global knowledge, and must be implemented, at least in part, by the operating system. Our work differs from that of Lu et al. by focusing on aggressive read-ahead and write-behind policies that can lead to bursty device-level access patterns even for interactive applications. Our work is more similar to that of Zeng et al., but without the notion of energy as a first-class resource. While we agree that energy awareness should be integrated into all aspects of the operating system, it is not clear to us that it makes sense to allocate joules to processes in the same way we allocate cycles or bytes. Rather than say “I’d like to devote 20% of my battery life to MP3 playback and 40% to emacs,” we suspect that users in an energy-constrained environment will say “I’d like to extend my battery life as long as possible without suffering more than a 20% drop in sound quality or interactive responsiveness.” It will then be the responsibility of the operating system to manage energy *across applications* to meet these quality-of-service constraints.

Recently, researchers have begun to explore methods to reduce the power consumption of large-scale storage systems. Carrera et al. [7] compare design schemes for conserving disk energy in network servers. Colarelli et

al. [2] explore massive arrays of idle disks, or MAID, as an alternative to conventional mass storage systems for scientific computing. Papathanasiou et al. [30] suggest replacing server-class disks with power-efficient arrays of laptop disks.

Disk access pattern “shaping” techniques, such as our prefetching algorithm, can be applied to server storage systems and improve their power efficiency. Zhu et al. [39] propose power-aware storage cache management algorithms that provide additional opportunities for a large-scale storage system to save energy. Our prefetching algorithm complements nicely their power-aware cache replacement policy. Finally, Gurumurthi et al. [13] suggest the use of DRPM [13], an approach that would dynamically modulate disk speed, decreasing the power required to keep the platters spinning when the load is light.

Prefetching. Prefetching has been suggested by several researchers as a method to decrease application perceived delays caused by the storage subsystem. Previous work has suggested the use of hints as a method to increase prefetching aggressiveness for workloads consisting of both single [31] and multiple [34] applications. Cao et al. [1] propose a two-level page replacement scheme that allows applications to control their own cache replacement, while the kernel controls the allocation of cache space among processes. Kaplan et al. [18] explore techniques to control the amount of memory dedicated to prefetching. Curewitz et al. [3] explore data compression techniques in order to increase the amount of prefetched data. To the best of our knowledge, previously proposed prefetching algorithms do not address improved energy efficiency. In general, they assume a non-congested disk subsystem, and they allow prefetching to proceed in a conservative way resulting in a relatively smooth disk usage pattern.

7 Conclusion

In our study we investigated page prefetching and caching strategies that increase the burstiness of I/O patterns in order to save energy in disks with non-operational low-power states. In addition, we presented methods to predict future accesses automatically and aggressively, to balance the memory requirements of prefetching and caching, and to coordinate accesses of concurrently running applications so that requests are generated and arrive at the disk at roughly the same time.

We have implemented our ideas in the Linux 2.4.20 kernel. Experiments with a variety of applications show that our techniques can increase the length of idle phases significantly compared to a standard Linux kernel leading to disk energy savings of 60–80%. The savings depend on the amount of available memory, and increase as the system’s memory size increases. Even relatively

short increases in the average idle interval length can lead to significant energy savings, mostly by making more efficient use of intermediate low-power states. Published studies [23, 5] attribute 9-32% of the total laptop energy consumption to the hard disk. These figures imply that our prefetching algorithm may increase battery lifetime by up to 25%. The exact fraction depends on the system configuration and the executing workload.

Though our current work has focused on hard disks, increased burstiness could be used to improve the energy efficiency of other devices with non-operational low-power states. Network interfaces—for wireless networks in particular—are an obvious example, but they introduce new complications. First, in addition to standby states, several wireless interfaces support multiple active states, with varying levels of broadcast power suitable for communication over varying distances. Second, while a disk never initiates communication, a wireless network does. Third, the energy consumed by a wireless interface depends on the quality of the channel, so communication bursts should be scheduled, when possible, during periods of high channel quality.

Over time, we speculate that burstiness may become important in the processor domain as well. In a processor with multiple clock domains, for example [33], one can save dynamic power in a floating-point application by slowing down the (lightly-used) integer unit. Alternatively, by scheduling instructions for burstiness, one might save both dynamic *and* static power by gating off voltage to the integer unit during periods of inactivity. This tradeoff between operational scale-down, with a smooth access pattern, and non-operational power-down, with a bursty access pattern, arises in multiple situations (including DRPM [13]), and is likely to be a recurring theme in our future work.

References

- [1] CAO, P., FELTEN, E. W., AND LI, K. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the 1995 ACM Joint Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'95/PERFORMANCE'95)* (1995), pp. 188–197.
- [2] COLARELLI, D., AND GRUNWALD, D. Massive Arrays of Idle Disks for Storage Archives. In *Proc. of the 2002 ACM/IEEE Conf. on Supercomputing (SC'02)* (Nov. 2002), pp. 1–11.
- [3] CUREWITZ, K. M., KRISHNAN, P., AND VITTER, J. S. Practical Prefetching via Data Compression. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'93)* (May 1993), pp. 257–266.
- [4] DOUGLIS, F., KRISHNAN, P., AND BERSHAD, B. Adaptive Disk Spin-down Policies for Mobile Computers. In *Proc. of the 2nd USENIX Symp. on Mobile and Location-Independent Computing* (Apr. 1995).
- [5] DOUGLIS, F., KRISHNAN, P., AND MARSH, B. Thwarting the Power-Hungry Disk. In *Proc. of the 1994 Winter USENIX Conf.* (Jan. 1994), pp. 293–306.
- [6] ELLIS, C. S. The Case for Higher Level Power Management. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems (HotOS VII)* (Mar. 1999).
- [7] ENRIQUE V. CARRERA, EDUARDO PINHEIRO, R. B. Conserving Disk Energy in Network Servers. In *Proc. of the 17th Annual ACM Intl. Conf. on Supercomputing (ICS'03)* (June 2003), pp. 86–97.
- [8] FERGUSON, G., AND ALLEN, J. TRIPS: An Intelligent Integrated Problem-Solving Assistant. In *Proc. of the 15th National Conf. on Artificial Intelligence (AAAI-98)* (July 1998), pp. 567–573.
- [9] FLAUTNER, K., AND MUDGE, T. Vertigo: Automatic Performance-Setting for Linux. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation (OSDI'02)* (Dec. 2002), pp. 105–116.
- [10] FLINN, J., AND SATYANARAYANAN, M. Energy-Aware Adaptation for Mobile Applications. In *Proc. of the 17th ACM Symp. on Operating Systems Principles* (Dec. 1999), pp. 48–63.
- [11] GOLDING, R., II, P. B., STAELIN, C., SULLIVAN, T., AND WILKES, J. Idleness is not sloth, Jan. 1995.
- [12] GOVIL, K., CHAN, E., AND WASSERMAN, H. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. In *Proc. of the 1st Annual Intl. Conf. on Mobile Computing and Networking (MobiCom'95)* (Nov. 1995).
- [13] GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., AND FRANKE, H. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proc. of the 30th Intl. Symp. on Computer Architecture (ISCA'03)* (June 2003), ACM Press, pp. 169–181.
- [14] HEATH, T., PINHEIRO, E., HOM, J., KREMER, U., AND BIANCHINI, R. Application Transformations for Energy and Performance-Aware Device Management. In *Proc. of the 11th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'02)* (Sept. 2002).
- [15] HELMBOLD, D. P., LONG, D. D. E., AND SHERROD, B. A Dynamic Disk Spin-down Technique for Mobile Computing. In *Proc. of the 2nd Annual Intl. Conf. on Mobile Computing and Networking (MobiCom'96)* (Nov. 1996).
- [16] HUANG, X., ALLEVA, F., HON, H.-W., HWANG, M.-Y., AND ROSENFELD, R. The SPHINX-II speech recognition system: an overview. *Computer Speech and Language* 7, 2 (1993), 137–148.
- [17] IBM Corporation. OEM Hard Disk Drive Specifications for DARA-2xxxxx (6 GB – 25 GB). 2.5-Inch Hard Disk Drive with ATA Interface. Revision (2.1), Nov. 1999.
- [18] KAPLAN, S. F., MCGEOCH, L. A., AND COLE, M. F. Adaptive Caching for Demand Prepaging. In *Proceedings of the 3rd Intl. Symp. on Memory Management* (2002), ACM Press, pp. 114–126.

- [19] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems* 10, 1 (Feb. 1992), 3–25.
- [20] KUENNING, G. H., AND POPEK, G. J. Automated Hoarding for Mobile Computers. In *Proc. of the 16th ACM Symp. on Operating Systems Principles* (Oct. 1997), ACM Press, pp. 264–275.
- [21] LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. S. Power Aware Page Allocation. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)* (Nov. 2000), pp. 105–116.
- [22] LI, K., KUMPF, R., HORTON, P., AND ANDERSON, T. Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proc. of the 1994 Winter USENIX Conf.* (Jan. 1994), pp. 279–291.
- [23] LORCH, J. R., AND SMITH, A. J. Energy consumption of Apple Macintosh computers. *IEEE Micro* 18, 6 (Nov. 1998), 54–63.
- [24] LU, Y.-H., BENINI, L., AND MICHELI, G. D. Power-Aware Operating Systems for Interactive Systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems* 10, 1 (Apr. 2002).
- [25] Micron Technology Inc. Micron 256Mb and 512Mb: x16 TwinDie SDRAM (Revision A 5/03 EN), May 2003.
- [26] MOGUL, J. C. A Better Update Policy. In *Proc. of the USENIX Summer 1994 Technical Conf.* (June 1994).
- [27] NOBLE, B., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile Application-Aware Adaptation for Mobility. In *Proc. of the 16th ACM Symp. on Operating Systems Principles* (Oct. 1997).
- [28] PAPATHANASIOU, A. E., AND SCOTT, M. L. Increasing Disk Burstiness for Energy Efficiency. Tech. Rep. 792, Computer Science Department, University of Rochester, Nov. 2002.
- [29] PAPATHANASIOU, A. E., AND SCOTT, M. L. Energy Efficiency Through Burstiness. In *Proc. of the 5th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'03)* (Oct. 2003), pp. 44–53.
- [30] PAPATHANASIOU, A. E., AND SCOTT, M. L. Power-efficient Server-class Performance from Arrays of Laptop Disks. Tech. Rep. 837, Computer Science Department, University of Rochester, Apr. 2004.
- [31] PATTERSON, R. H., GIBSON, G., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed Prefetching and Caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles* (Dec. 1995), pp. 79–95.
- [32] PERING, T., BURD, T., AND BRODERSEN, R. Voltage Scheduling in the lpARM Microprocessor System. In *Proc. of the 2000 Intl. Symp. on Low Power Electronics and Design (ISLPED'00)* (July 2000), pp. 96–101.
- [33] SEMERARO, G., MAGKLIS, G., BALASUBRAMONIAN, R., ALBONESI, D. H., DWARKADAS, S., AND SCOTT, M. L. Energy Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *Proc. of the 8th Intl. Symp. on High Performance Computer Architecture (HPCA-8)* (Feb. 2002), pp. 29–40.
- [34] TOMKINS, A., PATTERSON, R. H., AND GIBSON, G. Informed multi-process prefetching and caching. In *Proc. of the 1997 ACM Joint Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'97)* (1997), ACM Press, pp. 100–114.
- [35] WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. Scheduling for Reduced CPU Energy. In *Proc. of the 1st USENIX Symp. on Operating Systems Design and Implementation (OSDI'94)* (Nov. 1994).
- [36] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation (OSDI'02)* (Dec. 2002).
- [37] ZENG, H., FAN, X., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)* (Oct. 2002).
- [38] ZENG, H., FAN, X., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. Currency: Unifying Policies for Resource Management. In *Proc. of the USENIX 2003 Annual Technical Conf.* (June 2003).
- [39] ZHU, Q., DAVID, F., ZHOU, Y., DEVARAJ, C., AND CAO, P. Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management. In *Proc. of the 10th Intl. Symp. on High Performance Computer Architecture (HPCA-10)* (Feb. 2004).

Time-based Fairness Improves Performance in Multi-rate WLANs

Godfrey Tan and John Guttag

MIT Computer Science and Artificial Intelligence Laboratory
{godfreyt, guttag}@csail.mit.edu

Abstract

The performance seen by individual clients on a wireless local area network (WLAN) is heavily influenced by the manner in which wireless channel capacity is allocated. The popular MAC protocol DCF (Distributed Coordination Function) used in 802.11 networks provides equal long-term transmission opportunities to competing nodes when all nodes experience similar channel conditions. When similar-sized packets are also used, DCF leads to equal achieved throughputs (*throughput-based fairness*) among contending nodes.

Because of varying indoor channel conditions, the 802.11 standard supports multiple data transmission rates to exploit the trade-off between data rate and bit error rate. This leads to considerable *rate diversity*, particularly when the network is congested. Under such conditions, throughput-based fairness can lead to drastically reduced aggregate throughput.

In this paper, we argue the advantages of *time-based fairness*, in which each competing node receives an equal share of the wireless channel occupancy time. We demonstrate that this notion of fairness can lead to significant improvements in aggregate performance while still guaranteeing that no node receives worse channel access than it would in a single-rate WLAN. We also describe our algorithm, TBR (Time-based Regulator), which runs on the AP and works with any MAC protocol to provide time-based fairness by regulating packets. Through experiments, we show that our practical and backward compatible implementation of TBR in conjunction with an existing implementation of DCF achieves time-based fairness.

1 Introduction

802.11 is the *de facto* wireless networking standard. In a typical deployment, a mobile node or station equipped with an 802.11 interface communicates over the air to an access point (AP) or base station that is connected to a wired backbone. There are a number of different 802.11 standards. For concreteness, we focus primarily on 802.11b, the most widely used version of 802.11. When multiple mobile nodes wish to use the wireless channel simultaneously, the channel must be apportioned in some “fair” way among them. In 802.11 networks, the apportioning is controlled by DCF at the MAC layer and the queuing mechanism used at the APs.

For reasons we discuss later, nodes connected to 802.11 WLANs transfer data at a number of different rates. So, for example, the channel capacity might have to be apportioned between nodes transferring data at 11 Mbps and nodes transferring data at 1 Mbps. In this paper, we first demonstrate that DCF and the existing queuing schemes at the APs provide a notion of fairness that is inherently inefficient, and then propose and evaluate a better mechanism.

The signal strength and loss rate of indoor wireless channels vary widely, even for nodes that are equidistant from access points [19]. When the 802.11 MAC detects a packet loss (due to the absence of a synchronous *ack*), it continues retransmitting the packet until the maximum retry limit has been reached. However, this is futile when the average signal strength at the receiver is consistently lower than the threshold required for successful packet reception. In such cases, the sender can transmit at a lower data rate (using a more resilient modulation scheme) so that the channel bit error rate (BER) is reduced. In general, there is a trade-off between data rate and BER [11, 16].

Many vendors of APs and client cards implement automatic rate control schemes in which the sending stations adaptively change the data rate based on perceived channel conditions [7, 16, 21]. Many cards also allow users to manually set the data rate. The 802.11b standard defines four different data rates, 1, 2, 5.5 and 11 Mbps respectively. This leads to *rate diversity* in the system, where competing nodes within a cell use different data rates to communicate with the AP (in both up-link and downlink directions). As shown in Figure 1, various data transmission rates were used during 90-minute sessions of a student workshop that took place at MIT. Furthermore, WLANs carry significant amounts of traffic, and thus many APs experience several congested periods. In Section 3, we discuss the prevalence of rate diversity in more detail.

When multiple nodes are simultaneously exchanging data using different data rates during congested periods, the total network throughput is quite different from what one might expect. Figure 2 illustrates how the aggregate throughput can be dramatically reduced when two competing nodes use different data rates to upload files using TCP. The achieved throughput of the node with the higher transmission rate is reduced by about 3.75 times.

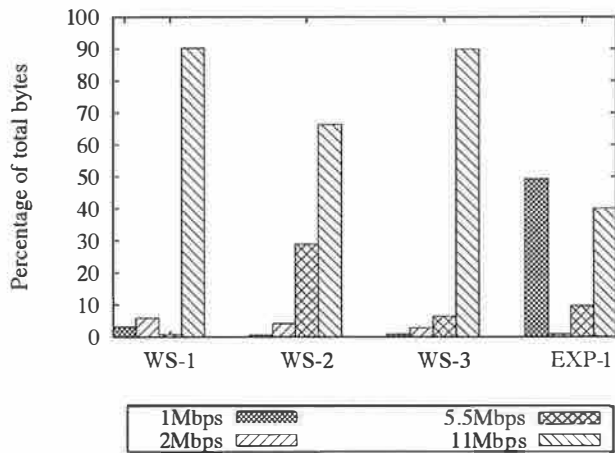


Figure 1: Fractions of bytes transferred at various data rates during three 90-minute workshop sessions (WS) and an experiment (EXP-1).

The root cause of this behavior is the definition of fairness used by DCF. This variant of the CSMA medium access protocol is designed to give approximately equal *transmission opportunities* to each competing node. That is to say each node will have approximately the same number of opportunities to send a data frame, *irrespective of the amount of time required to transmit a packet*. When the same-sized packets are used and channel conditions are similar, each competing node, regardless of its data rate, achieves roughly the same throughput, as shown in Figure 2.

Since the node transmitting at 1 Mbps will take several times longer to transmit a frame than the node transmitting at 11 Mbps, the channel is being used most of the time by the slower node. In Figure 2, the fraction of the channel time used by the slower node is 6.4 times as much as that used by the faster node. Hence, the total throughput is reduced to a level much closer to what one gets when both competing nodes are slow. The faster node pays a penalty for competing against a slow node rather than against another fast node.

Aggregate throughput is also impacted. Naively, one might expect the total throughput of an 11 Mbps and a 1 Mbps channel to be somewhere around 2.93 Mbps, the average of the total throughputs achieved by a pair of 11 Mbps channels (5.08 Mbps) and a pair of 1 Mbps channels (0.78 Mbps). However, it is only 1.34 Mbps, less than half of what one might expect. And the situation is likely to become worse as the emerging 802.11g networks, with a maximum data rate of 54 Mbps, are deployed alongside relatively slower 802.11b networks. 802.11g users may see far less performance improvement than expected, thus lowering the incentive for users to upgrade to 802.11g cards.

DCF mainly affects the channel capacity allocation in the up-link direction. The packet scheduling mechanism at the AP

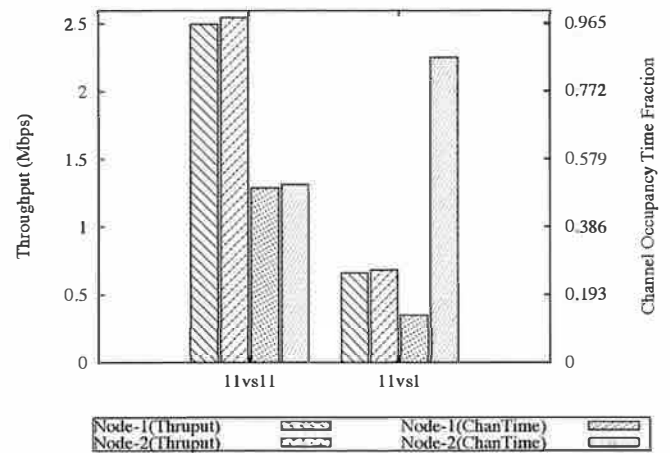


Figure 2: TCP throughputs achieved and fractions of channel occupancy time used by two competing nodes when i) both sending at 11 Mbps and ii) one sending at 11 Mbps and the other at 1 Mbps.

dictates the channel capacity allocation to clients in the down-link direction. When there are multiple backlogged packets destined to more than one clients, the scheduling scheme must decide the order of transmission. Again, since the channel conditions at the clients vary, different data transmission rates are often used for different clients. Scheduling schemes in the literature [8, 9, 24] provide throughput-based fairness that has been widely-accepted in wired networks and single-rate 802.11 WLAN, in which the data rate for each transmission on the shared medium is the same. When such scheduling schemes are employed at the APs of multi-rate WLANs, the channel capacity allocation on the downlink direction is impacted in a similar undesirable way as in the uplink direction.

We believe that these inefficiencies are best addressed by adopting a notion of fairness that gives each competing client node an approximately equal amount of the shared channel resource: *channel occupancy time*. This notion of *Time-based* fairness is quite different from the throughput-based fairness notion widely accepted in wired networks and single-rate WLANs. Time-based fairness provides an important property in multi-rate WLANs that throughput-based fairness does not:

Baseline property: The long-term throughput of a node competing against any number of nodes running at different speeds is equal to the throughput that the node would achieve in an existing single-rate 802.11 WLAN in which all competing nodes were running at its rate.

I.e., the throughput a node achieves when competing against n nodes is identical to what it would achieve if it were competing against n nodes all using its data rate.

Fairness is, of course, a subjective notion (as any parent of

multiple children knows). We do not claim that one notion is “fairer” than the other. However, we do point out that in the presence of rate diversity during congested periods, time-based fairness does improve the overall network performance.

In this paper, we:

- Examine the impact of both time-based and throughput-based fairness on various measures of network efficiency
- Present an analytic framework in which the impact of rate diversity on the network performance is quantitatively evaluated for each fairness notion used
- Validate our model against a deployed 802.11b network
- Show, by collecting and analyzing trace data, that current 802.11b networks indeed suffer the predicted performance degradation in the presence of rate diversity
- Present an effective and efficient scheme, TBR (for Time-based Regulator), for deploying time-based fairness in existing AP-based WLANs, irrespective of the MAC protocol used
- Describe an efficient 802.11-based implementation of TBR that requires changing only the driver on the access point, and
- Demonstrate the relative advantage of time-based fairness, both analytically (using our model) and experimentally (using the 802.11-based implementation)

The rest of this paper is organized as follows. Section 2 analyzes the expected performance impact of both notions of fairness and examines which notion of fairness DCF achieves under various circumstances. Section 3 presents network trace analyses and experiments that demonstrate that rate diversity is common in today’s networks. Section 4 describes in detail our scheme to achieve the time-based fairness, Section 5 evaluates our scheme’s performance and Section 6 discusses related work.

2 Analysis

In this section, we argue why time-based fairness is desirable in some cases and analyze the achieved throughputs of competing nodes, possibly using different data rates and packet sizes, in 802.11-like CSMA WLANs.

2.1 Impact of Fairness Notions on Efficiency

We now examine how different notions of fairness impact the overall efficiency of multi-rate WLANs. The measure of fairness between nodes i and j with equal priorities during an interval (t_1, t_2) is: $|\alpha_i(t_1, t_2) - \alpha_j(t_1, t_2)|$, where $\alpha_i(t_1, t_2)$ and $\alpha_j(t_1, t_2)$ are their achieved portions of the shared resource. In this paper, we only focus on nodes with equal priorities. Different notions of fairness are captured by differing definitions

of α . Let $\alpha_i^t(t_1, t_2)$ and $\alpha_i^r(t_1, t_2)$ be the channel occupancy time and the achieved throughput respectively of node i during (t_1, t_2) .

The choice of fairness notion dictates how the network allocates the shared resource (in our case channel capacity) during periods in which demand exceeds supply. The overall network performance as well as the performance of individual nodes can be greatly affected by it. We define network efficiency as the sum of the utility of each competing node based on their shares of shared resource. We use two traffic models, a *fluid model* [8, 27] and a *task model* [4], to examine the impact of fairness notions on overall network efficiency.

In the fluid model, there is a finite number of flows, each of which continuously transfers infinite streams of bits. The network efficiency can be evaluated using its (average) aggregate sustained throughput (*AggrThroughput*). Note that while the instantaneous throughput of a node will vary depending upon its data rate, the expected instantaneous total throughput is time invariant.

In the task model, there is a finite number of flows, each of which transfers a finite number of bits. Since we are providing fairness only among competing nodes, we assume that each node has one flow. In this model, the instantaneous aggregate throughput varies with the remaining task mix. Thus, it is more appropriate to look at network efficiency in other ways such as the average task completion time, *AvgTaskTime*, and the final task completion time, *FinalTaskTime*. Short *AvgTaskTime* is especially desirable for mobile nodes since those that have completed their communication tasks can turn-off their wireless cards to save energy or move to another place to go on with their work. Short *FinalTaskTime* is also desirable since it implies higher long term average aggregate throughput and thus the network can potentially accommodate more tasks.

Figures 3(a) and 3(b) compare the achieved TCP throughputs and the channel occupancy times of two competing nodes when different fairness notions (RF and TF) are used. These figures assume a flow model or a task model in which no flow has yet completed. The graphs are based on the experiments we conducted. In the remainder of this section we demonstrate that these experimental results are consistent with analytical predictions.

Observe that when both nodes transmit at the same rate (11vs11 and 1vs1), the allocations of both throughputs and channel occupancy times are identical for both fairness notions. However, when one node ($n1$) transmits at 1 Mbps and the other ($n2$) at 11 Mbps (see middle bars in figures), nodes achieve equal throughputs under throughput-based fairness, but $n2$ achieves more throughput than $n1$ under time-based fairness. The situation is reversed with respect to the allocation of channel occupancy time. Each node achieves an equal amount of channel occupancy time under time-based fairness, but $n1$ gets a much larger share than $n2$ under throughput-based fairness.

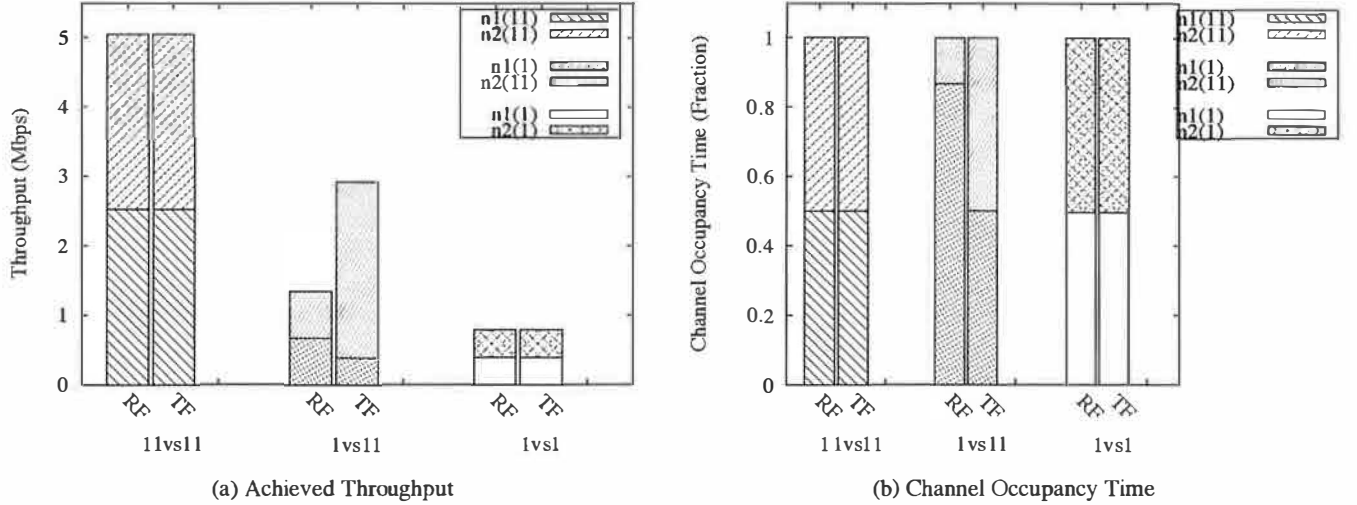


Figure 3: Achieved TCP throughputs and fractions of channel occupancy time of two competing nodes in three different combinations of data rates: 11 vs 11, 1 vs 11 and 1 vs 1. *throughput-based fairness* and *TF* denote the throughput-based and time-based fairness notions respectively. E.g. in 3(a), $n1(11)$ denotes the throughput achieved by node $n1$ transmitting at 11 Mbps.

Criteria	Measure	RF	TF
Fairness	$ \alpha_i^r(t_1, t_2) - \alpha_j^r(t_1, t_2) $	Better	Worse
	$ \alpha_i^t(t_1, t_2) - \alpha_j^t(t_1, t_2) $	Worse	Better
Efficiency (task model)	<i>FinalTaskTime</i>	Same	Same
	<i>AvgTaskTime</i>	Worse	Better
Efficiency (fluid model)	<i>AggrThruput</i>	Worse	Better

Table 1: Comparison of different measures when the throughput-based (RF) and time-based (TF) fairness notions are enforced.

Compared to throughput-based fairness, time-based fairness benefits faster nodes at the expense of slower nodes. However, the fairness property captured by the *baseline property* of Section 1 is maintained. Each class of node performs as it would in a single-rate 802.11 WLAN. For instance, the achieved throughput of $n1$ in both 1vs11 and 1vs1 cases is the same under time-based fairness. The same statement can be made for other performance measures such as per-packet latency.

Table 1 compares various measures of fairness and efficiency for scenarios in which nodes within a cell compete using different data rates. As explained in the rest of this section, the conclusions captured in this table hold for any number of nodes. However, for concreteness, we use the 1vs11 case as a concrete example. Under the task model, we assume that each node has an equal amount of data to transfer. Technically, the same results apply so long as each node has a similar distribution of task size.

When the fluid traffic model is used, higher *AggrThruput* results under time-based fairness as evident in Figure 3(a). *FinalTaskTime* remains unchanged under the task model since the network is work-conserving under both fairness notions. However, *AvgTaskTime* under time-based fairness is lower than that under throughput-based fairness. Under throughput-based fairness, $AvgTaskTime = FinalTaskTime$, since both tasks complete at the same time. Under time-based fairness, in contrast, $AvgTaskTime < FinalTaskTime$. This is because the task of the 11 Mbps node will complete sooner, since it achieves higher throughput while the completion time of the 1 Mbps node remains the same.

The rest of this section examines how well existing 802.11's DCF achieves each notion of fairness, and presents an analytical framework to predict the network performance in multi-rate WLANs.

2.2 Fairness in AP-based WLANs

Traditional fair queuing algorithms designed for wired networks attempt to provide a fair allocation of the bandwidth on a shared link [8, 9, 24]. Previous work on Fair scheduling in wireless networks generally adopted this notion of fairness [20, 22, 27]. However, unlike wired links, typical wireless networks are half-duplexed in that the channel needs to be shared for both transmitting and receiving packets.

In AP-based WLANs, each AP is just a facilitator and thus the resource used by it to transmit packets destined to a client should be accounted as part of the resource used by the client or its flow. In the rest of this paper, we focus on providing fair channel time shares among competing nodes, not flows. The channel time used by a competing node is the total chan-

nel time used in both transmitting and receiving packets to and from the AP. We believe that this notion is more intuitive than the traditional notion of providing fair resource allocations among competing flows. The latter is more suitable for wired networks and *ad hoc* wireless networks, where there are no facilitators present (e.g. when the medium is shared by nodes in a distributed manner) or the facilitator is the only one transmitting on the medium (e.g. router scheduling packets to transmit on an output link).

2.3 Network Model

In this subsection, we describe the network model that we use to analyze the performance of AP-based WLANs. In these infrastructure-based WLANs, each wireless node only communicates directly with an AP in order to exchange data with another node inside or outside of the WLAN.

As in much of the existing literature [8, 27], we base our analysis on the fluid traffic model, and thus are concerned with *AggrThroughput*. However, the results in this section clearly indicate that when the task traffic model is used, the network efficiency in terms of *AvgTaskTime* is better under time-based fairness than under throughput-based fairness (see Section 2.1).

Let I be the set of competing nodes and n its cardinality. We define d_i and s_i as the data rate used and data packet size used by node i . For simplicity of analysis, we assume that d_i and s_i apply to data packets in both uplink and downlink directions of node i .

We define the *channel occupancy time* $T(i)$, $0 \leq T(i) \leq 1$, of node i as the fraction of time a wireless node i is able to access the channel to either transmit or receive packets to and from the AP. The channel occupancy time necessary to transfer a data packet includes i) the transmission time of the data packet, ii) the transmission time of a synchronous *ack*, iii) the propagation delays, iv) the inter-frame idle periods necessary for a node to be idle before accessing the channel, and v) the amount of time required to perform retransmissions when necessary. Since we assume that the channel is busy all the time:

$$\sum_{i \in I} T(i) = 1 \quad (1)$$

Let $R(I)$ and $R(i)$ be the total throughput achieved by all nodes in I and the achieved throughput of node i respectively. We can express $R(i)$ in terms of $T(i)$ as:

$$R(i) = T(i) \times \gamma(d_i, s_i, I) \quad (2)$$

where $\gamma(d_i, s_i, I)$ is the *baseline throughput* (that nodes experience) as a function of d_i , the data rate, and s_i , the packet size, holding all else equal. The baseline throughput $\gamma(d_i, s_i, I)$ equals the maximum total achieved throughput when all nodes (I) use the same packet size and data rate under similar loss characteristics. For instance, when two nodes simultaneously transfer files using 1500-byte TCP packets and a data rate of

11 Mbps, the baseline throughput (as shown in Figure 2) is 5.08 Mbps. However, the actual throughput $R(i)$ node i depends upon the fraction of time i was able to access the channel, $T(i)$. The total actual throughput of the network is simply:

$$R(I) = \sum_{i \in I} R(i) \quad (3)$$

Baseline throughput increases with the increase in data transmission rate as well as packet size. The latter is due to reduced per-packet overhead as a result of the larger number of payload bits per packet. By expressing $R(i)$ in terms of $\gamma(d_i, s_i, I)$, we avoid dealing directly with other factors that affect the throughput such as the back-off periods and physical layer overhead, that are independent of the work covered in this paper. $\gamma(d, s, I)$ can be obtained both theoretically and experimentally. In Section 2.7, we report measured values of $\gamma(d, s, I)$ for various values of d . Furthermore, we do not deal with varying loss characteristics since our goal is in understanding how diverse data rates and packet sizes affect the network performance.

2.4 Impact of DCF on Fairness Notions

802.11's DCF (Distributed Coordinating Function) is far-and-away the most commonly used contention resolution method in 802.11 networks. Although an alternative Point Coordinating Function (PCF) exists, it is not implemented by most AP vendors because of its complexity and issues of co-existence with DCF-based networks. DCF gives equal transmission opportunities (or long-term channel access probability) to each contender [17, 26].

Therefore, competing nodes attempting to send data packets to the AP over the same time interval will be able to transmit equal numbers of frames. DCF's transmission opportunity based mechanism provides fair allocations of both throughput and channel occupancy time only if all contending nodes i) use the same data rate, ii) use the same packet size, and iii) experience very similar loss characteristics. If only the last two conditions hold, DCF achieves throughput-based fairness but does not achieve time-based fairness. For any other combination, DCF achieves neither time-based fairness nor throughput-based fairness.

Figure 4 shows the throughputs achieved by three competing nodes that are either sending or receiving data using the maximum data rate of 11 Mbps and the maximum packet size of 1500 bytes. In uplink directions, the throughput achieved by each node is approximately equal due to DCF. In downlink directions, the throughput achieved by each node is approximately equal largely due to the AP queuing scheme, which usually transmits to wireless clients in a round-robin manner. TCP throughputs are significantly less than UDP throughputs because the transmission overhead of TCP *ack* packets. The total throughputs achieved in the uplink direction are higher than those in the downlink direction. This is because one 802.11

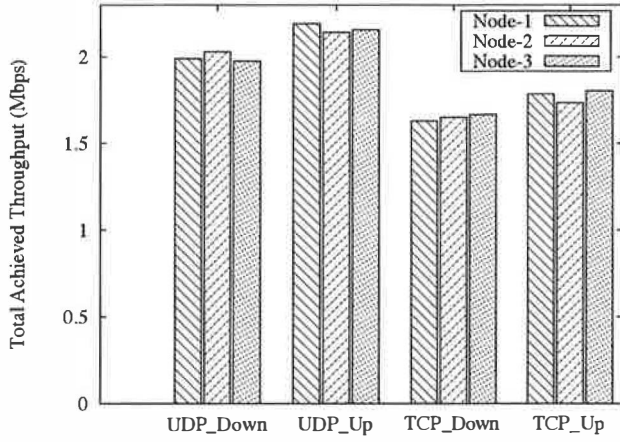


Figure 4: UDP and TCP throughputs achieved by three competing nodes (Cisco-350 cards) each of which is exchanging data at 11 Mbps with a common AP (Cabletron Roamabout-2000). “Up” and “Down” x-axis labels denote that the nodes are sending data to and receiving from the AP respectively.

sending node (the AP) cannot fully utilize or saturate the channel since a transmitting node is required to back-off for a random period, between 0 and 610 us, after every successful packet transmission. This overhead is reduced with the increase in number of competing nodes.

We now derive the general expression of $T(i)$, the fraction of time node i is able to transmit or receive packets under DCF. For ease of notation, we will use γ_i in place of $\gamma(d_i, s_i, I)$. For steady state performance, we can assume that in each round, each competing node transfers a single packet. Thus, $T(i)$ is simply the ratio of the time required for node i to transfer a data frame, which is $\frac{s_i}{\gamma_i}$, to the total time required for every node in I to transfer a data frame.

$$T(i) = \frac{\frac{s_i}{\gamma_i}}{\sum_{j \in I} \frac{s_j}{\gamma_j}} \quad (4)$$

2.4.1 Impact of Rate Diversity

To understand the impact of rate diversity, let's assume that each node uses the same packet size, i.e. $\forall i, j \in I, s_i = s_j$. Therefore, based on Equations 2 and 3,

$$T(i) = \frac{\frac{1}{\gamma_i}}{\sum_{j \in I} \frac{1}{\gamma_j}} \quad (5)$$

$$R(i) = \frac{1}{\sum_{j \in I} \frac{1}{\gamma_j}} \quad (6)$$

$$R(I) = \frac{n}{\sum_{j \in I} \frac{1}{\gamma_j}} \quad (7)$$

Equation 6 clearly shows that the throughput of each node i is the same. Thus, under these conditions, DCF achieves

throughput-based fairness. Observe, however, the amount of throughput is dependent on the baseline throughputs of all nodes in I , which in turn depend on their data rates and packet sizes.

The channel occupancy time $T(i)$ of node i is inversely proportional to the baseline throughput of node i , which increases with the increase in transmission rate. Thus, as expected, nodes with slower data rates occupy the channel much longer than those with higher data rates, leading to degradation in the overall network performance.

2.4.2 Impact of Packet Size Diversity

The impact of packet size diversity can be understood by assuming that each node uses the same data rate, i.e. $\forall i, j \in I, d_i = d_j$. Based on Equations 2 and 3, we have:

$$T(i) = \frac{\frac{s_i}{\gamma_i}}{\sum_{j \in I} \frac{s_j}{\gamma_j}} \quad (8)$$

$$R(i) = \frac{s_i}{\sum_{j \in I} \frac{s_j}{\gamma_j}} \quad (9)$$

$$R(I) = \frac{\sum_{i \in I} s_i}{\sum_{j \in I} \frac{s_j}{\gamma_j}} \quad (10)$$

Once again, $R(i)$ depends on the baseline throughputs of all other competing nodes. However, the equations make it clear that in this case $T(i)$ and $R(i)$ may differ across nodes, depending upon packet size.

2.5 Impact of the AP Queuing Scheme

The queuing mechanism at the AP dictates the channel bandwidth allocation to clients in the downlink direction. Since the channel conditions at the clients vary, different data transmission rates are often used for different clients. As far as we know, the existing literature on scheduling schemes [20, 22, 27] does not consider the impact of rate diversity. Thus, the aggregate network throughput when only downlink traffic is present is impacted in the same way as previously explained. We also note that if loss rates experienced by nodes differ and both packet transmissions in uplink and downlink directions use different data rates, the achieved throughputs of competing clients may not be equal or easily predictable, even when all nodes use DCF and the AP employs a fair queuing scheme.

2.6 Impact of Time-based Fairness

Under the time-based fairness, our proposed definition of fairness, each node achieves an equal share of channel occupancy time. Thus,

$$T'(i) = \frac{1}{n} \quad (11)$$

Substituting Equation 11 in Equations 2 and 3,

$$R'(i) = \frac{\gamma_i}{n} \quad (12)$$

d (Mbps)	s (Byte)	$n = I $	$\gamma(d, s, I)$
11	1500	2	5.189
5.5	1500	2	3.327
2	1500	2	1.493
1	1500	2	0.806

Table 2: The experimentally achieved total throughput (or the baseline throughput) of the two nodes simultaneously exchanging data at the same data rate d and packet size s . Each node has a similar frame loss rate of less than 2%.

Fairness Criteria	$R(n1)$ (1)	$R(n2)$ (2)	$R(n3)$ (11)	$R(n4)$ (11)	Total
RF	0.436	0.436	0.436	0.436	1.742
TF	0.202	0.373	1.30	1.30	3.175

Table 3: Comparison of achieved throughputs (in Mbps) of four nodes, each transmitting at 1, 2, 11 and 11 Mbps respectively, under RF and TF. Note that $R(n1)$ under TF is the same as what $n1$ would achieve if all $n2$, $n3$ and $n4$ transmit at 1 Mbps.

$$R'(I) = \frac{1}{n} \sum_{i \in I} \gamma_i \quad (13)$$

Notice that $R'(i)$ only depends on what node i can achieve under the given conditions and the number of competing nodes. It does not depend upon the data rates or packet sizes used by competing nodes. Unlike $R(I)$ shown in previous subsections, $R'(I)$ is a simple summation of each node's maximum achievable throughput when all competing nodes use its data rate and packet size. $R'(I)$, and $R(I)$ in Equations 7 and 10 will be equal if and only if all nodes in I use the same data rate and packet size.

2.7 Examples

In this section we illustrate the ramifications of the differences between Equation 12 and Equation 6 with a small example.

Table 2 shows the experimentally derived baseline throughputs of two identical competing nodes as a function of transmission rate. This provides an estimate of baseline throughput for various transmission rates.

Using these values, we compute the throughputs when I contains four competing nodes, one communicating at 1 Mbps, one at 2 Mbps, and at 11 Mbps. These are shown in Table 3. The achieved throughput of the slower nodes is less under time-based fairness than under throughput-based fairness. Under time-based fairness, the 1 Mbps and 2 Mbps nodes achieve the throughput they would have achieved if all four nodes were running at their speed. The 11 Mbps nodes achieve considerably higher throughput under time-based fairness, and the total throughput improves by 82%.

3 Existence of Rate Diversity

In this section, we discuss in detail i) whether rate diversity exists in today's 802.11b networks and ii) whether a single user or multiple users are actively exchanging data during the intervals in which the network is saturated.

To investigate the prevalence of rate diversity, we collected traces of wireless network traffic at one-day Iris student workshop at MIT. There were about 45 attendees and more than half turned on their wireless laptops. We set up a laptop to sniff data during each of the three 90-minute sessions, WS-1, WS-2 and WS-3, all of which took place in a single room of about 40' \times 25'.

Figure 1 shows the fractions of data bytes transferred using each of the four possible rates during each session. It is clear that rate diversity exists even in a relatively small room. During WS-2, more than 30% of the data bytes were transferred using data rates lower than 11 Mbps.

We also set up an experiment to investigate how an AP change data rates to various clients in indoor office environments. We placed a Cabletron Roamabout-2000 AP in a 18' \times 14' office 7' above ground. A sender with a wired connection to the AP sent unicast UDP data packets at the saturation rate simultaneously to four different receivers. The first node was about 4' away from the AP, the second 12' and one thin, wooden wall away, the third 26' and two thin wooden walls away and the fourth 30' and two thick walls in between. As shown in Figure 1 (see EXP-1), more than 50% of the bytes were transferred using the lowest data rate.

In fact, a recent extensive wireless network usage study on a university campus has found that the average received signal strength varies widely even among positions that are within 20' of an access point [19]. Thus, we believe that rate diversity is prevalent in many indoor WLANs and its impact would be much more pronounced with mixed deployments of 802.11b and 802.11g networks.

The negative impact of rate diversity is significant only if the following two conditions are true: i) more than one competing node exchange data during the periods in which network is saturated and ii) competing nodes use diverse data rates. Our analysis of this particular workshop trace data, however, shows that the network is well over-provisioned with 7 APs providing a combined channel capacity of 33 Mbps. However, recent studies have shown that in many enterprise networks [2] and university residential halls [18], WLANs carry significant traffic and contain many APs that have a lot of busy or congested periods.

We analyzed wireless tcpdump trace of Whittemore, a residential facility in the Dartmouth business school where students were required to own laptops. This data was collected by Kotz et al. over the Spring semester [18] and was made publicly available by Kotz. Unfortunately, the trace data does not contain the data transmission rate used for each frame transmis-

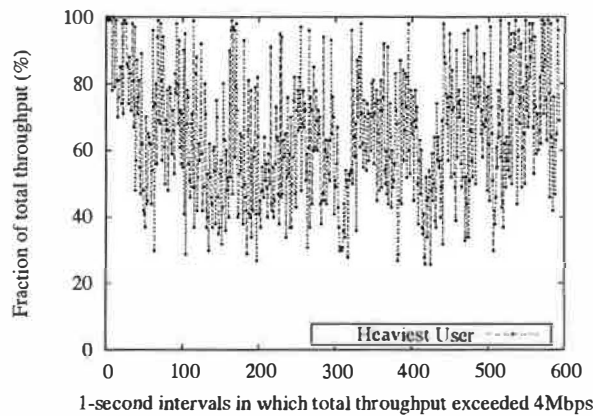


Figure 5: The fraction of throughput achieved by the heaviest user at a busy AP during busy 1-second intervals.

sion. Nonetheless, we can identify the busy periods in which an AP is carrying close-to the maximum amount of data, and investigate whether more than one user actively exchange data during congested periods.

Since TCP dominated the traffic, we conservatively define *busy or congested intervals* as those in which the total data throughput at the AP exceeded 4 Mbps, 80% of the commonly observed TCP saturation throughput when nodes transmit at the maximum data rate and experience a very low loss rate of 1% to 2%.

Figure 5 plots the fraction of aggregate throughput achieved during busy 1-second intervals by the heaviest user at an AP at Whittemore on 8 April 2002, a Spring Monday. The heaviest user is one that exchanged the most bytes with the AP. Although the majority of bytes were transferred by one user on average, it is clear that the heaviest user alone rarely saturated the channel. In most 1-second busy intervals, users other than the heaviest user exchanged significant amounts of data.

4 Time-based Regulator

In the previous sections, we have argued that competing nodes should be given an equal amount of long-term channel occupancy time. As explained before, in AP-based WLANs, the MAC protocol and the queuing scheme at the AP in combination determine the channel time allocation. Therefore, to achieve a desired channel time allocation, coordination is necessary between the MAC protocol and the queuing scheme. Our proposed Time-based Regulator runs at each AP, coordinates with clients when necessary and works in conjunction with any MAC protocol.

TBR provides an equal share of long-term channel occupancy time to each competing client node by

- Dictating how packet transmissions are scheduled at the AP as well as at the clients

```

PROCEDURE ASSOCIATEEVENT( $i$ ) {
     $tokens_i \leftarrow T_{init}$ 
     $bucket_i \leftarrow T_{init}$ 
     $rate_i \leftarrow$  fair share of channel occupancy time
    initialize  $queue_i$ 
}
PROCEDURE FILLEVENT( $t$ ) {
    for each  $bucket_i$ 
         $tokens_i \leftarrow tokens_i + (t * rate_i)$ 
        if ( $tokens_i > bucket_i$ )
             $tokens_i \leftarrow bucket_i$ 
}
PROCEDURE APPTXEVENT( $p$ ) {
     $i \leftarrow$  destination of  $p$ 
    enqueue  $p$  to  $queue_i$ 
}
PROCEDURE MACTXEVENT() {
    for each station  $i$  starting with  $next_i$ 
        if  $queue_i$  is not empty and  $tokens_i > 0$ 
            dequeue a packet  $p$  from  $queue_i$ 
            ask the MAC to transmit  $p$ 
             $next_i \leftarrow$  next station after  $i$ 
}
PROCEDURE COMPLETEEVENT( $p$ ) {
     $t \leftarrow$  channel occupancy time of  $p$ 
    if  $p$  was sent by AP
         $i \leftarrow$  destination of  $p$ 
    else
         $i \leftarrow$  source of  $p$ 
     $tokens_i \leftarrow tokens_i - t$ 
    if ( $actual_i = 0$ )
         $start_i \leftarrow$  current time
     $actual_i \leftarrow actual_i + t$ 
}

```

Figure 6: Pseudo-code of TBR

- Taking into account the channel occupancy time of traffic in both downlink and uplink directions, and
- Taking into account varying traffic conditions, loss rates, data rates, and frame sizes

A typical implementation of TBR requires no modification to the underlying MAC protocol and to the drivers of mobile clients, allowing incremental deployment and preserving backward compatibility. Modifications to the clients, however, are necessary to preserve correctness in cases where the uplink UDP flows make up a significant fraction of the WLAN traffic. We will discuss more on this issue in the next subsection.

TBR is based on the leaky bucket scheme [3]. The fundamental unit or token used in the implementation is the channel occupancy time in terms of micro-seconds. TBR only schedules the

transmission of a packet destined to or originated from a client only if the node has not used up all its available channel time.

Figure 6 shows the pseudo-code of TBR that runs on the AP. TBR sits above the MAC layer and below the network layer and is implemented in five event handlers, each of which is triggered by the upper layer, timer or the MAC layer.

When a node i associates with the AP (i.e. joins the network), ASSOCIATEEVENT is triggered. The procedure i) creates output queue $queue_i$ and ii) initializes $tokens_i$, the available tokens, $bucket_i$, the maximum amount of tokens that the node can accumulate, and $rate_i$, the rate at which tokens are being re-filled.

Whenever the upper layer has a packet p to transmit, it calls APPTXEVENT. TBR simply enqueues the packet to $queue_i$ where i is the destination of p .

TBR adjusts $tokens_i$ according to the channel occupancy time of transmitted frames originated from or destined to node i . Section 4.2 described how TBR computes the channel occupancy time. $bucket_i$ determines the maximum length of the burst period in which node i can transmit successively (if no other nodes can transmit). $bucket_i$ can affect the short-term fairness and we discuss this issue later in Section 4.5.

TBR sets up a timer that periodically calls FILLEVENT, which for each node i , updates $tokens_i$ according to $rate_i$ and t , the time elapsed since the last time FILLEVENT was called. $rate_i$ is the rate at which tokens are being re-filled. We note that $\sum_{i=1}^n rate_i = 1$, where n is the number of active client nodes. In general, $rate_i$ can vary among client nodes depending on the desired fairness policy. If each competing node should receive an equal share of the channel occupancy time, $rate_i = \frac{1}{n}$. However, in practice, not all nodes can consume their available channel time according to the allocation. TBR ensures that the system remain work conserving by adjusting the token rates appropriately as discussed in Section 4.3.

4.1 Scheduling Frame Transmissions

Whenever the MAC layer is ready to accept a new packet for transmission, it calls HWTXEVENT. TBR decides which backlogged packet to release as follows. TBR chooses one output queue among all the output queues with positive available channel time (tokens) and dequeues a packet for transmission.

The manner in which the output queue is chosen has no impact on the overall correctness since only the queues with positive tokens are considered. Nonetheless, the order could impact the short-term fairness. For simplicity and to alleviate short-term unfairness, TBR chooses the output queue among those with positive tokens in a round-robin manner. We note that short-term unfairness can further be reduced by choosing the queue which has the packet with the shortest potential final completion time as in traditional fair queuing schemes [8, 24].

Once the output queue is chosen, TBR can decide which frame in the queue gets transmitted. For TCP, in-order packet deliv-

ery is desirable and thus first-in-first-out discipline is preferable. However, if there are time-sensitive packets (used by real-time protocols), they should have priority over TCP packets with earlier arrival times. The correctness of TBR does not depend on how a packet to dequeue is chosen. We also note that TBR works with any buffering scheme (e.g. RED, drop-tail), whose goal is to decide which packets to drop when the queue is getting full. Note that we distinguish buffering schemes from packet scheduling schemes. The former is responsible for deciding which packets to drop whereas the latter decides which packet gets transmitted [8].

TBR also dictates the scheduling of packet transmissions at the clients. Specifically, whenever $tokens_i \leq 0$, TBR needs to explicitly inform node i to delay transmission for a short amount of time. This can be accomplished in two major ways. First, the TBR agent at the AP informs the client by either sending an explicit notification packet or piggyback such information in a downlink packet when possible. Second, the client monitors the total channel occupancy time of packets transmitted and received and transmits only if there is available channel time allocated for the node. To do so, the client only needs to know $rate_i$. However, as we explain in Section 4.3, TBR at the AP may update $rate_i$ depending on the overall traffic conditions and when that happens, TBR needs to inform the client. In both cases, a client agent is necessary at each client to communicate with TBR at the AP. We choose the first method for simplicity.

The actual amount of communication overhead depends on the MAC protocol used. TBR requires a single bit in the MAC header of a data frame transmission to inform the client to delay its transmission for a pre-determined amount of time. In cases where there is only uplink traffic, TBR can still use the same procedure if the underlying MAC protocol (e.g. DCF) employs a *stop-and-go* retransmission strategy. A stop-and-go protocol requires the node receiving a data frame to reply with a synchronous acknowledgment, which can carry the TBR notification bit. Furthermore, if the underlying MAC protocol employs a polling mechanism (such as 802.11's PCF), no explicit communication is necessary since TBR can dictate which node gets polled.

Cooperation from each client is only necessary if the client has uplink UDP flows that represent a significant fraction of its traffic. Studies of WLAN traffic at university campuses [18, 25] and at a multi-day conference [1] show that TCP accounted for more than 90% of bytes exchanged over the WLANs. TCP data packets are paced by TCP *ack* packets ("ack clocking" [13]) sent out by the receiver. In a typical scenario, all TCP data and *ack* packets go through the same AP. Therefore, delaying TCP data (*ack*) packets at the AP has the effect of slowing down the sending rates of downlink (uplink) TCP flows.

We note that our current TBR implementation does not contain the client-side implementation of TBR. As we demonstrate in Section 5, TBR without the client cooperation can effectively

provide long-term channel time guarantees for TCP flows in both directions as well as downlink UDP flows.

4.2 Computing Channel Occupancy Time

Whenever the MAC layer has either finished sending or received packet p , it triggers `COMPLETEEVENT`. This procedure subtracts the channel occupancy time of p from the tokens associated with node i that is the source or destination of p . It also modifies $actual_i$, the actual tokens used since $start_i$. We will explain how TBR uses $actual_i$ in the next subsection.

We now describe how to compute the channel occupancy time for packet p . We define *packet transfer time* as the total time required to transfer a data packet at the 802.11 MAC layer, which is typically the sum of i) the transmission time of the data packet, ii) the transmission time of a synchronous MAC-layer *ack* when necessary, iii) propagation delays for both the data and *ack* packets, and iv) the inter-frame idle periods necessary for the sending node to be idle before accessing the channel. Since the MAC-layer may perform retransmissions upon a transmission failure, the channel occupancy time is the sum of the packet transfer time of each transmission until p has successfully been transmitted or dropped as a result of an undeliverable failure. Therefore, failed packets also contribute to the channel occupancy time of the sending node.

Taking into account retransmissions is straight forward in the downlink direction. However, in the uplink direction, the AP is not aware of the exact number of retransmission attempts made by the client stations. Ideally, the underlying MAC protocol should include a retry sequence number field (about 4 bits) in the header to indicate how many retransmissions precede the current packet transmission.

When retransmission information is not available for each packet received and the necessary header modification is not an option, the AP needs to estimate the information necessary to compute the channel occupancy time. We distinguish two types of losses at the AP: one detected at the MAC layer (due to the CRC check failure) and the other at the physical layer. In the former, it is highly likely that the MAC header, whose size is relatively much smaller than the typical payload size, is not corrupted and thus the AP can determine the source address of the failed transmission as well as the transmission rate. We note that the MAC layer header can be made robust against channel errors by transmitting at a lower data rate.

However, if the frame loss is detected at the physical layer, TBR can be aware of the loss but may not know the necessary transmission information. We believe that heuristics can be developed to estimate the transmission information of each loss detected at the physical layer based on i) the number of active clients in the last few dozen milliseconds, ii) the likelihood of each client contending, and iii) their steady state loss rates at the downlink direction. We plan to develop such heuristics in the future.

4.3 Keeping Channel Utilization High

When traffic contains a mixture of TCP and UDP flows that have various sending rates (and bottleneck link bandwidth), it is important to correctly determine the amount of channel occupancy time made available to each node. Specifically, TBR needs to adjust $rate_i$ to reflect changing traffic conditions. For instance, the system will be under-utilized if we give each node $\frac{1}{n}$ of the available channel time but some nodes cannot consume all of their available time shares whereas others can consume more if allowed.

TBR periodically adjusts $rate_i$ associated with each node i so that the channel utilization is kept at maximum without violating the max-min fairness constraint [6, 14]. That is, the smallest $rate_i$ in the network must be as large as possible. Subject to this constraint, the second smallest token rate must also be as large as possible.

We note that DCF in conjunction with a simple round-robin queuing scheme at the AP generally achieves the max-min notion of fairness when only TCP flows are involved. Assume that there are 3 uplink TCP flows and that one flow can only consume $\frac{1}{5}$ of the channel bandwidth (the wireless hop is not its bottleneck link). DCF will allow each of the remaining flows to consume $\frac{2}{5}$ of channel bandwidth provided that the bottleneck link of both flows is the wireless link.

TBR with any MAC protocol achieves the same fairness criteria provided that the MAC layer has the work conserving property that DCF does, i.e. each client node with data to transmit contends for channel access opportunistically. Notice that the max-min fairness criteria does not require that the actual demand of each node is known. Rather, one can simply achieve the fairness goal by incrementally giving more channel time to each competing node that can consume all the channel time made available to it [3]. We implement this general idea in TBR.

Initially each competing node starts with the desired token rate of $\frac{1}{n}$. TBR schedules a timer event called `ADJUSTRATEEVENT` that periodically adjusts the token *rate* available to each node. As shown in Figure 7, `ADJUSTRATEEVENT` computes the excess capacity of the *under-utilized nodes*, each with the actual token rate ($actual_i$) lower than the assigned rate by the threshold R^{th} . It then computes the excess capacity E^{min} to redistribute equally among nodes (I') that have fully utilized the provisioned bandwidth in the previous round.

The actual method of computing E^{min} is of little importance for the long-term correctness so long as E^{min} is not too big. However, E^{min} does affect the responsiveness of TBR to changing traffic conditions. We will discuss more about this in Section 4.5. If E^{min} is too large, the instantaneous throughputs experienced by flows can significantly vary. Such behaviors may increase the buffer requirements at the nodes to avoid TCP *ack* compression that can lead to packet drops.

Figure 7 shows a particular way of choosing E^{min} . We pick,

```

PROCEDURE ADJUSTRATEEVENT() {
  for each node  $i$ 
     $excess \leftarrow rate_i - \frac{actual_i}{now-start_i}$ 
    if ( $excess \leq R^{th}$ )
      if  $excess < E^{min}$ 
         $E^{min} \leftarrow d$ 
      if  $excess > E^{max}$ 
         $m \leftarrow i$ 
    else
      add  $i$  to set  $I'$ 
       $E^{min} \leftarrow E^{min} \div 2$ 
    for each node  $j \in I'$ 
       $rate_j \leftarrow rate_j + \frac{E^{min}}{|I'|}$ 
     $rate_m \leftarrow rate_m - E^{min}$ 
    for each node  $j \in I$ 
       $actual_j \leftarrow 0$ 
}

```

Figure 7: Pseudo-code of the token rate adjustment event

among all under-utilized nodes, node m with the maximal excess capacity (the largest difference in actual and assigned token rate). Half of E^{min} is subtracted from m 's token rate and the other half redistributed among nodes that have consumed tokens at rates close to their assigned rates. In Section 5, we show that TBR is able to keep the channel utilization high in the presence of varying traffic conditions.

4.4 An 802.11-based Implementation

We implemented TBR in the HostAP [15] driver running on a Linux PC as a proof of concept. The HostAP driver implements access point functionality so that PCs equipped with popular Prism chipset based 802.11 cards can act as APs. We use unique 6-byte MAC addresses as node identifiers.

We note that TBR requires APs to set up per-node output queue. However, the total buffer space requirement is comparable between a normal AP and an AP with TBR. For instance, if an existing AP has the total queue size of x packets than a TBR-equipped AP can setup n queues each with $\frac{x}{n}$ packets, where n is the number of competing nodes. For ease of implementation, our TBR implementation uses FIFO queues. As explained before, TBR can work with any buffering scheme.

Finally, we note that the current implementation of TBR does not use the retransmission information in computing the packet transfer time but we plan to do so in the future. Thus, TBR in some cases can cause slight biases in granting channel occupancy time to competing nodes. Nonetheless, as we show in Section 5, it does well in achieving its goal.

4.5 Discussion

TBR is currently intended for ensuring that each competing node receives an equal share of channel occupancy time based on max-min fairness over the long run. As we later demonstrate in Section 5, TBR works well when competing flows last for hundreds of packets.

Although we believe that long-lived flows (e.g. file transfer applications) are usually the cause of congestion in enterprise and university networks, we acknowledge that congestion in *hot-spot* access networks may be caused by many short-lived flows with diverse data rates, each sending only dozens of packets.

Responsiveness of TBR relies on how it adjusts the token rate assigned to each competing node and how often (see ADJUSTRATEEVENT). Furthermore, the burst period ($bucket_i$) in which node i can transmit successively also influences the responsiveness of TBR as well as short-term fairness. Special attention must be paid to a packet-level interaction between TBR and the underlying MAC so that TBR can respond to varying traffic conditions in the order of tens of packet transfer time. In the future, we plan to understand each of these issues in detail and make TBR responsive for very short-lived flows as well.

Large $bucket_i$ can exacerbate the short-term unfairness, i.e. some competing nodes do not achieve their desired fair shares within a very short interval, commonly found in 802.11 WLANs [17]. Short-term unfairness in its most severe form leads to TCP ack compression in which multiple TCP acks arrive at the sender, which then sends several TCP packets successively, leading to undesirable packet drops at the bottleneck queue. However, the TCP ack compression problem can be effectively solved by pacing TCP packets [5].

TBR can potentially be modified to provide each competing node with the desired share of channel occupancy time (not necessarily equal). Therefore, QoS mechanisms may use TBR to provide QoS at existing AP-based WLANs. We also note that although the current implementation of TBR allocates channel time to nodes, it can be extended to allocate channel time among various flows of each node.

We note that the 802.11e standard [12] currently being drafted defines quality of service support for the 802.11 MAC. Using 802.11e, competing nodes acquire Transmission Opportunities (TXOP), each of which is defined as an interval of time when a station has the right to initiate transmissions. TXOPs are allocated via contention or granted through the centralized coordinator like the AP. 802.11e differentiates the probability of channel access based on the traffic categories. TBR can be integrated with 802.11e by choosing appropriate traffic categories for each competing node according to their fair share of channel occupancy time.

5 Evaluation

We setup experiments to evaluate the correctness and performance of TBR. We used a PIII-700MHz Linux laptop

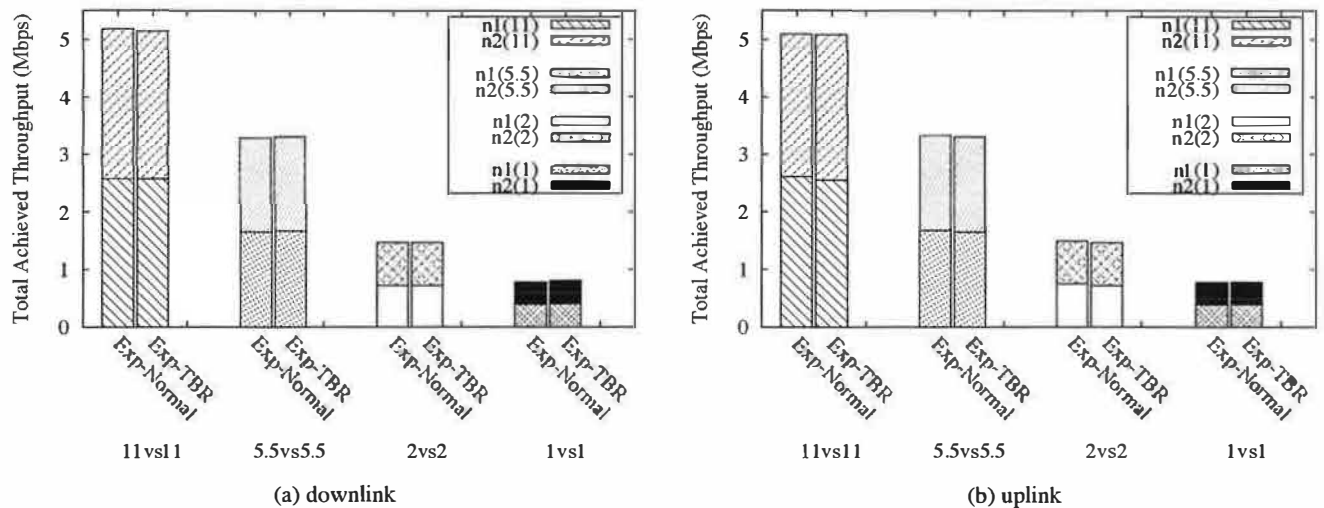


Figure 8: TCP throughputs achieved in either uplink or downlink direction by two competing nodes using the same data rate. *Exp-Normal* and *Exp-TBR* denote the experiments that were run with the AP equipped without or with TBR respectively. *n1(11)* denotes the throughput achieved by node *n1* transmitting at 11 Mbps.

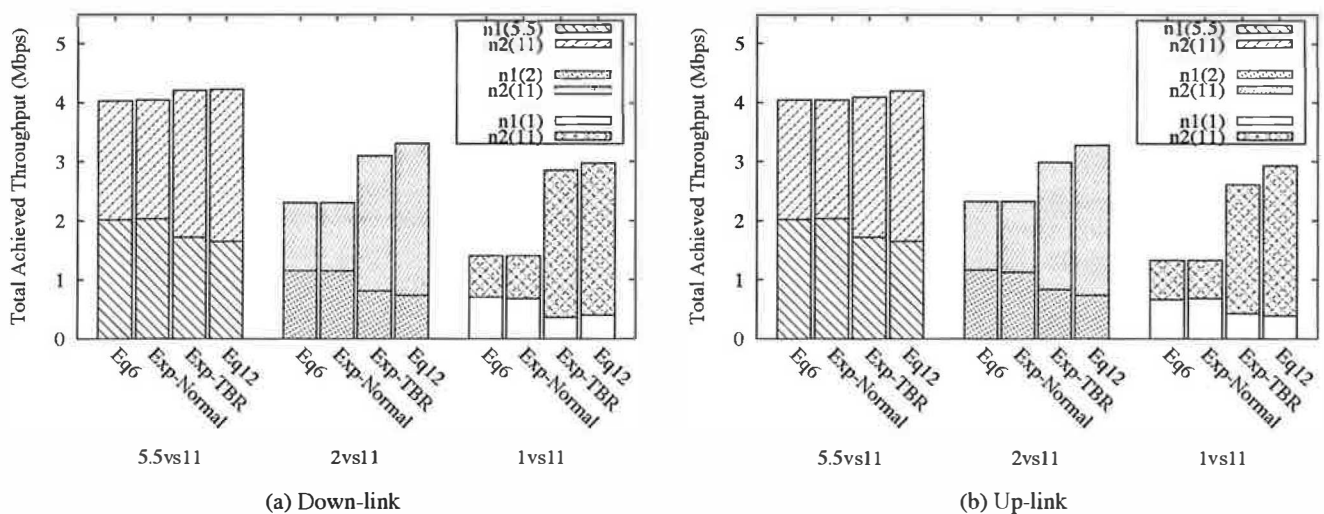


Figure 9: TCP throughputs achieved in either up-link or down-link direction by two competing nodes using different data rates. *Exp-Normal* and *Exp-TBR* denote the experiments that were run with the AP equipped without or with TBR respectively. *Eq6* and *Eq12* represent the achieved throughputs according to Equation 6 and Equation 12 respectively. *n1(11)* denotes the throughput achieved by node *n1* transmitting at 11Mbps.

equipped with a D-Link DWL-650 card running the Hostap driver as the AP and IPAs equipped with Cisco-350 cards as competing nodes.

For each type of experiment, we ran in two different AP configurations: one with TBR, *Exp-TBR*, and one without, *Exp-Normal*. Each data point is an average of 5 to 10 runs and in each run, each contending node sends about 2000 1500-byte packets. All throughputs measured are achieved TCP through-

puts.

When the AP is run under the normal configuration, no queue is set up in the driver. Instead, the kernel interface queue (with the maximum size of 110) is used to store packets. When the AP is run with TBR, *n* queues each with the maximum queue size of $\frac{100}{n}$ is set up inside the driver. The kernel interface queue is then set to 10. Thus, the total buffer space available to each scheme is the same.

Figure 8 compares the throughputs achieved by two competing nodes when the AP is configured with or without TBR. When competing nodes use the same data rate, *Exp-TBR* and *Exp-Normal* yield almost identical results, showing that TBR incurs little overhead.

When nodes use different data rates, the throughput achieved by each competing node as well as the total throughput differ significantly depending upon whether TBR is used or not. As shown in Figure 9(a), when TBR is used, the total achieved throughput in the down-link direction increases by about 6% in the 5.5vs11 case, 35% in the 2vs11 case and 103% in the 1vs11 case.

Analytical (*Eq6*) and experimental (*Exp-Normal*) values agree for all the cases when the AP is configured without TBR. Similarly, *Exp-TBR* and *Eq12* show very similar results, affirming that our regulator achieves the objective of providing long-term equal channel occupancy time to competing nodes. The slight differences in performance between *Exp-TBR* and *Eq12* is due to the fact that TBR needs to estimate channel occupancy time without the retransmission information available. Whenever a packet loss is experienced by a node, the channel occupancy time of that node needs to be decreased accordingly. Without the retransmission information, TBR in this case slightly biased the node sending at a lower data rate, thus decreasing the total throughput by a small amount compared to *Eq12*. In the future, we plan to extract (from the card firmware) or estimate retransmission information as suggested in Section 4.

Figure 9(b) shows similar improvements achieved by TBR in the up-link direction. We also ran experiments involving mixed up-link and down-link TCP flows and found similar results (not shown here).

Throughput	Exp-Normal	Exp-TBR
n1	2.9434	2.9542
n2	2.1276	2.1193
Total	5.071	5.061

Table 4: Comparison of achieved TCP throughputs under *Exp-Normal* and *Exp-TBR*. Node *n2* experienced the bottleneck bandwidth of 2.1 Mbps whereas node *n1* could send as fast as it could (TCP permitted). Both nodes transmitted at 11 Mbps.

To understand how well TBR works when traffic contains flows with various demands, we set up a scenario that involved two nodes, *n1* and *n2*, each sending TCP packets at the same data rate of 11 Mbps but experienced different bottleneck link capacities. *n2* experienced the bottleneck bandwidth of 2.1 Mbps while the wireless link is *n2*'s bottleneck. We achieved this by limiting the sending rate of the application generating TCP packets at *n2*. The expected DCF's behavior is to give *n2* 2.1 Mbps of channel bandwidth and *n1* the remainder. Table 4 shows the throughputs achieved under *Exp-TBR* and *Exp-*

Normal. There is no significant difference between the two sets of results showing that the rate adjustment algorithm described in Section 4.3 works.

6 Related Work

We note that the general idea of temporal sharing in the context of multi-rate WLANs has been mentioned before by Sadeghi *et al.* [23]. They have proposed an opportunistic rate adaptation scheme (called OAR) that achieves significant throughput gain over previously proposed rate adaptation schemes [11, 16]. The key idea behind OAR is to allow nodes that have high-quality channel condition to transmit more than one packet at a time taking advantage of time-correlated channel conditions. OAR simply allows a node that can transmit at 11 Mbps 5 times more opportunities than the node transmitting at 2 Mbps. OAR justifies this by saying that nodes are achieving similar time-shares as when they both are transmitting at 2 Mbps. OAR is a DCF-based protocol mainly intended for *ad hoc* networks and requires modifications to DCF. Unlike AP-based networks, *ad hoc* networks, in which nodes communicate with each other without using access points, are more suitable when communications among wireless nodes are dominant or no wired infrastructure exists. In contrast, AP-based networks are designed for communications among wireless nodes and other nodes that can be reached via a wired infrastructure to which APs are connected.

Unlike the previous work, we investigate and explain the differing impacts of the fairness notions on the network performance and our work focuses on AP-based 802.11 networks in which the queuing scheme at the AP significantly impacts the channel capacity allocation.

Recently, Heusse *et al.* have shown through simulations and experiments that performance degradation occurs when two nodes are sending at different data rates [10]. Through analysis, authors show that the node sending at a lower data rate will achieve the same throughput as other nodes sending at higher data rate. The authors do not suggest any mechanism to mitigate this effects.

Efforts have been made in developing distributed fair scheduling algorithms that are suitable for the shared wireless medium. [20, 22, 27]. Like the schemes proposed in wired networks [8, 9, 24], these wireless scheduling algorithms [20, 22, 27] neither take into account the impact of transmission rate diversity nor the channel resource for both downlink and uplink traffic as most schemes [22, 27] were targeted for *ad hoc* wireless networks.

7 Summary and Conclusion

We started by showing that, in the presence of rate diversity, the throughput-based fairness notion implemented by the 802.11's popular MAC protocol and the traditional queuing schemes at the APs leads to a situation in which the aggregate throughput is determined largely by the slowest node.

We next presented a time-based notion of fairness that provides an equal amount of long-term channel occupancy time to each competing node. This prevents faster nodes from being dragged down by slower ones. Moreover, it satisfies what we called the *baseline property*, i.e., the achieved throughput of any competing node in a multi-rate WLAN is equal to what it would achieve in a single-rate WLAN in which all competing nodes transmit at its data rate. In the presence of rate diversity, using this definition of fairness can lead to vastly improved aggregate network throughput, more than 100% in some realistic scenarios.

We next described a practical scheme called TBR that works in conjunction with any MAC protocol to provide long-term time-based fairness in AP-based WLANs by appropriately scheduling packet transmissions. We showed that TBR can be implemented in an AP driver in a way that is backwards compatible with existing 802.11 standard. We implemented our scheme in the Linux Hostap driver running on a PC used as the AP, and evaluated it through a series of experiments. In the absence of rate diversity, the performance of our implementation is equivalent to the standard implementation. In the presence of rate diversity, it achieves the predicted gains.

In today's AP-based 802.11b WLANs, rate diversity is already common as our trace analyses show. As newer standards such as 802.11g are deployed, the problem will become worse. For an extended period of time 802.11 WLANs will run in a mixed mode, and if 802.11g clients are slowed down to run at the rate of 802.11b clients, there will be little incentive to upgrade. We believe that switching to time-based fairness is a good option.

References

- [1] A. Balachandran, G. M. Voelker, P. Bahl, and P. V. Rangan. Characterizing user behavior and network performance in a public wireless LAN. *ACM Press*, June 2002.
- [2] M. Balazinska and P. Castro. Characterizing mobility and network usage in a corporate wireless local-area network. In *Proc. of ACM MOBISYS'03*, May 2003.
- [3] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition, 1992.
- [4] J. Bruno, E. G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, Jul 1974.
- [5] M. C. Chan and R. Ramjee. TCP/IP performance over 3g wireless links with rate and delay variation. In *Proc. of ACM MOBICOM'02*, pages 71–82, 2002.
- [6] D.-M. Chiu and R. Jain. Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17(1):1–14, June 1989.
- [7] Data Sheet of Cisco Aironet 350 Series Access Points. http://www.cisco.com/warp/public/cc/pd/witc/ao350ap/prodlit/carto_in.htm.
- [8] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *Internetworking: Research And Experience*, 1:3–26, April 1990.
- [9] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, oct 1997.
- [10] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda. Performance anomaly of 802.11b. In *Proc. of IEEE INFOCOM'03*, April 2003.
- [11] G. Holland, N. H. Vaidya, and P. Bahl. A rate-adaptive MAC protocol for multi-hop wireless networks. In *Proc. of ACM MOBICOM'01*, pages 236–251, 2001.
- [12] IEEE 802.11 Working Group. Draft Supplement to International Standard for Information Exchange between systems - LAN/MAN Specific Requirements, Nov. 2001.
- [13] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review*, 18, 4:314–329, 1988.
- [14] R. Jain, D.-M. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System. Technical Report 301, Digital Equipment Corporation, Sept. 1984.
- [15] Jouni Malinen. Host AP driver for Intersil Prism2/2.5/3. <http://hostap.epitest.fi>, 2003. Version 0.0.1.
- [16] A. Kamerman and L. Monteban. Wavelan ii: A high-performance wireless lan for the unlicensed band. *Bell Labs Technical Journal*, pages 118–133, Summer 1997.
- [17] C. E. Koksal, H. I. Kassab, and H. Balakrishnan. An analysis of short-term fairness in wireless media access protocols. In *Proc. of ACM SIGMETRICS'00*, June 2000.
- [18] D. Kotz and K. Essien. Analysis of a campus-wide wireless network. In *Proc. of ACM MOBICOM'02*. ACM Press, Sept. 2002.
- [19] D. Kotz, C. Newport, and C. Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Dept. of Computer Science, Dartmouth College, July 2003.
- [20] S. Lu, V. Bharghavan, and R. Srikant. Fair scheduling in wireless packet networks. *IEEE/ACM Transactions on Networking*, 7(4):473–489, 1999.
- [21] ORiNOCO AS-2000 System Release Note. http://www.michiganwireless.org/tools/Lucent/ORiNOCO/AS-2000_Rel2_1/AS2000_R2_10_01_Readme.txt.
- [22] P. Ramanathan and P. Agrawal. Adapting packet fair queueing algorithms to wireless networks. In *Proc. of ACM MOBICOM'98*, pages 1–9, 1998.
- [23] B. Sadeghi, V. Kanodia, A. Sabharwal, and E. Knightly. Opportunistic media access for multirate ad hoc networks. In *Proc. of ACM MOBICOM'02*, sept 2002.
- [24] M. Shreedhar and G. Varghese. Efficient Fair Queueing using Deficit Round Robin. In *Proc. of ACM SIGCOMM'95*, August 1995.
- [25] D. Tang and M. Baker. Analysis of a metropolitan-area wireless network. *Wireless Networks*, 8(2/3):107–120, 2002.
- [26] Y. Tay and K. Chua. A capacity analysis for the IEEE 802.11 MAC protocol. *ACM/Baltzer Wireless Networks*, 7(2):159–171, Mar 2001.
- [27] N. H. Vaidya, P. Bahl, and S. Gupta. Distributed fair scheduling in a wireless LAN. In *Proc. of ACM MOBICOM'00*, pages 167–178, 2000.

EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks

Lewis Girod Jeremy Elson Alberto Cerpa
Thanos Stathopoulos Nithya Ramanathan Deborah Estrin

*Center for Embedded Networked Sensing
University of California, Los Angeles
Los Angeles, CA 90095 USA*

{girod,jelson,cerpa,thanos,nithya,destrin}@cs.ucla.edu

Abstract

Many Wireless Sensor Network (WSN) applications are composed of a mixture of deployed devices with varying capabilities, from extremely constrained 8-bit “Motes” to less resource-constrained 32-bit “Microservers”. EmStar is a software environment for developing and deploying complex WSN applications on networks of 32-bit embedded Microserver platforms, and integrating with networks of Motes. EmStar consists of **libraries** that implement message-passing IPC primitives, **tools** that support simulation, emulation, and visualization of live systems, both real and simulated, and **services** that support networking, sensing, and time synchronization. While EmStar’s design has favored ease of use and modularity over efficiency, the resulting increase in overhead has not been an impediment to any of our current projects.

1 Introduction

The field of wireless sensor networks (WSNs) is growing in importance [1], with new applications appearing in the commercial, scientific, and military spheres, and an evolving family of platforms and hardware. One of the most promising signs in the field is a growing involvement by researchers *outside* the networking systems field who are bringing new application needs to the table. A recent NSF Workshop report [4] details a number of these needs, building on early experience with deployments (e.g. GDI [7], CENS [23], James Reserve [26]).

Many of these applications lead to “tiered architecture” designs, in which the system is composed of a mixture of platforms with different costs, capabilities and energy budgets [5] [21]. Low capability nodes, often Crossbow Mica Motes [24] running TinyOS [17], can perform simple tasks and provide long life at low cost. The high capability nodes, or *Microservers*, generally consume more energy, but in turn can run more complex software and support more sophisticated sensors. EmStar is a software environment targeted at Microserver platforms.

Microservers, typically iPAQ or Crossbow Stargate platforms, are central to several new applications at CENS. The Extensible Sensing System (ESS) employs Microservers as

data sinks to collect and report microclimate data at the James Reserve. A proposed 50-node seismic network will use Stargates to measure and report seismic activity using a high-precision multichannel Analog to Digital Converter (ADC). Ongoing research in acoustic sensing uses iPAQ hardware to do beamforming and animal call detection. Although EmStar systems do not target Motes as a platform, EmStar systems can easily interoperate with Motes and Mote networks.

In this paper, we intend to show how EmStar addresses the needs of WSN applications. To motivate this discussion, Figure 1 details a hypothetical application for which EmStar is well-suited. In this example, several nodes collaborate to acoustically localize an animal based on its call—an improved version of our system described in [8]. The large dashed box shows how the system might be implemented by combining existing EmStar components (gray boxes) with hypothetical application-specific components (light gray dashed boxes). Because EmStar systems are composed from small reusable components, it is easy to plug new application-specific components into many different layers of the system.

Although most of the implemented components in the diagram are described in more detail later in the paper, we will briefly introduce them here. The `emrun` module serves as a management and watchdog process, starting up, monitoring, and shutting down the system. The `emproxy` module is a gateway to a debugging and visualization system. The `udpd`, `linkstats`, `neighbors` and `MicroDiffusion` modules implement a network stack designed to work in the context of wireless links characterized by highly variable link quality and network topology. The `timehist`, `syncd`, and `audiod` modules together implement an audio sampling service that supports accurate correlation of time series across a set of nodes. The hypothetical modules include `FFT`, which computes a streaming Fourier transform of the acoustic input, `detect`, which is designed to detect a particular acoustic signature, and `collab_detect`, which orchestrates collaborative detection across several nodes.

This application demonstrates several of the attributes that are special to WSNs. First, the nodes in the system

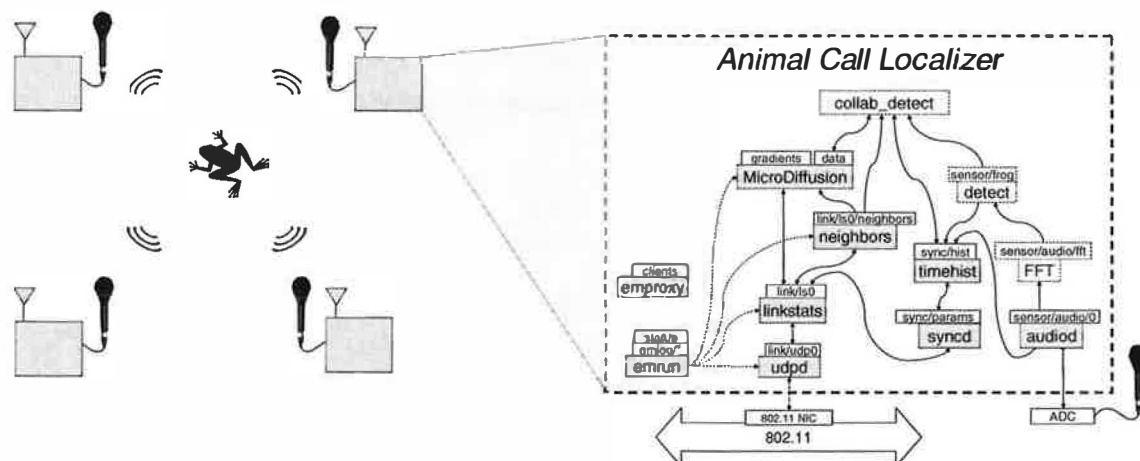


Figure 1: To motivate the EmStar design, we show a block diagram of a hypothetical WSN application to which EmStar is well suited. The diagram shows an improved version of our prototype animal call localization system described in [8]. In the new design, a network of “Localizer” nodes collaborate over a wireless network to localize an animal by its call. Each node detects the specific audio signature of the target animal and then collaboratively locates the target by comparing the arrival time of the signal at multiple points. The dashed box is an exploded view showing how EmStar components might be used to implement the Localizer nodes. The gray boxes represent existing EmStar modules, while the light gray dashed boxes represent hypothetical application specific modules. The white boxes represent various types of named device interface, including Sensor Devices, Link Devices, and Status Devices. Arrows indicate client-server relationships. Although all services have a control channel to EmRun, only four are shown, represented by dashed arcs.

have a higher probability of failure or disconnection than many Internet-based systems. Wireless connectivity and network topology can vary greatly, and systems deployed “in the wild” are also subject to hardware failures with higher probability. While Internet distributed systems often have low standards of client reliability, they typically assume a “core” of high reliability components that is not always present in a WSN.

Second, the digital signal processing (DSP) algorithms running on each node are complex and must work for a broad set of inputs that is difficult to characterize. In practice, this means that certain unexpected conditions may cause unforeseen error conditions. Fault tolerance and layers of filtering are needed to absorb these transients.

Third, energy considerations, along with aforementioned properties of wireless, influence the design of networking primitives. These issues favor soft state and hop-by-hop protocols over end-to-end abstractions. Energy considerations may also necessitate system-wide coordination to duty cycle the node. While many of these issues are similar to those addressed by TinyOS [17], EmStar is better suited to applications built on higher performance platforms.

2 Tools and Services

EmStar incorporates many tools and services germane to the creation of WSN applications. In this section, we briefly describe these tools and services, without much implementation detail. In Section 3, we detail key building blocks used to implement these tools. Then, in Section 4 we show how the implementation makes use of the building blocks.

2.1 EmStar Tools

EmStar tools include support for deployment, simulation, emulation, and visualization of live systems, both real and simulated.

EmSim/EmCee Transparent simulation at varying levels of accuracy is crucial for building and deploying large systems [9] [11]. Together, EmSim and EmCee comprise several accuracy regimes. EmSim runs many virtual nodes in parallel, in a pure simulation environment that models radio and sensor channels. EmCee runs the EmSim core, but provides an interface to real low-power radios instead of a modeled channel. The array of radio transceivers used by EmCee is shown in Figure 2(b).

These simulation regimes speed development and debugging; pure simulation helps to get the code logically correct, while emulation in the field helps to understand environmental dynamics before a real deployment. Simulation and emulation do not eliminate the need to debug a deployed system, but they do tend to reduce it.

In all of these regimes, the EmStar source code and configuration files are identical to those in a deployed system, making it painless to transition among them during development and debugging. This also eliminates accidental code differences that can arise when running in simulation requires modifications. Other “real-code” simulation environments include TOSSim [11] and SimOS [20].

EmView/EmProxy EmView is a graphical visualizer for EmStar systems. Figure 2(a) shows a screen-shot of EmView displaying real-time state of a running emulation. Through an extensible design, developers can easily add “plugins” for new applications and services. EmView uses

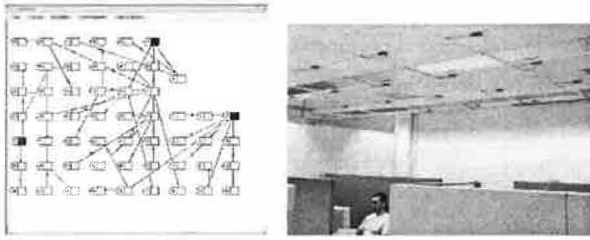


Figure 2: (a) EmView and (b) the Ceiling Array

a UDP protocol to request status updates from real or simulated nodes. Although the protocol is only best-effort, the responses are delivered with low latency, such that EmView captures real-time system dynamics. EmProxy is a server that runs on a node or as part of a simulation, and handles requests from EmView. Based on the request, EmProxy will monitor node status and report back changes in real time.

EmRun EmRun starts, stops, and manages running services in EmStar. It processes a config file that specifies how the EmStar services are “wired” together, and starts the system up in dependency order, maximizing parallelism. EmRun also maintains a control channel to each child process that enables it to monitor process health (respawn dead or stuck processes), initiate graceful shutdown, and receive notification when starting up that initialization is complete. Log messages emitted by EmStar services are processed centrally by EmRun and exposed to interactive clients as in-memory log rings with runtime-configurable loglevels.

2.2 EmStar Services

EmStar services include support for networking, sensing, and time synchronization.

Link and Neighborhood Estimation Wireless channels have a significant “gray zone” where connectivity is unreliable and highly time-varying [6]. Node failures are also common. Therefore, applications are brittle when they assume the topology is pre-configured. Dynamic neighbor discovery is a basic service needed by all collaborative applications if they are to be robust. Potential collaborators must be discovered at *run-time*.

EmStar’s Neighbors service monitors links and provides applications with a list of active, reliable nodes. Applications are notified when the list changes so that they can take action in response to environmental changes. The LinkStats service goes one step further: in exchange for slightly more packet overhead, it provides much finer-grained reliability statistics. This can be useful, for example, to a routing algorithm that weights its path choices by link reliability.

Time Synchronization The ability to relate the times of events on different nodes is critical to most distributed sensing applications, especially those interested in correlation of high-frequency phenomena. The TimeSync service pro-

vides a mechanism for converting among CPU clocks (i.e. `gettimeofday()`) on neighboring nodes. Rather than attempt to synchronize the clocks to a specific “master”, TimeSync estimates conversion parameters that enable a timestamp from one node to be interpreted on another node. Timesync can also compute relations between the local CPU clock and other clocks in the system, such as sample indices from an ADC or the clocks of other processor modules [3].

Routing EmStar supports several types of routing: Flooding, Geographical, Quad-Tree, and Diffusion. One of the founding principles of EmStar is that innovation in routing and hybrid transport/routing protocols are key research areas in the development of wireless sensor network systems. EmStar “supports” several routing protocols, but it also makes it easy to invent your own. For example, the authors of Directed Diffusion [16] [18] have ported diffusion to run on top of EmStar.

2.3 EmStar Device Support

EmStar includes native support for a number of devices, including sensors and radio hardware.

HostMote and MoteNIC EmStar systems often need to act as a gateway to a network of low-energy platforms such as Mica Motes running TinyOS. The HostMote service implements a serial line protocol between a Mote and an EmStar node. HostMote provides an interface to configure the attached Mote and an interface that demultiplexes Mote traffic to multiple clients. MoteNIC is a packet relay service built over HostMote. MoteNIC provides a standard EmStar data link interface, and pipes the traffic to software on the attached Mote that relays those packets onto the air.

Audio Server The Audio service provides buffered and continuous streaming interfaces to audio data sampled by sound hardware. Applications can use the Audio service to acquire historical data from specific times, or to receive a stream of data as it arrives. Through integration with the TimeSync service, an application can relate a specific series of samples on one node to a series taken at the same time on another node. The ability to acquire historical data is crucial to implementing triggering and collaboration algorithms where there may be a significant nondeterministic delay in communication due to channel contention, multihop communication, duty cycling, and other sources of delay.

3 Building Blocks

In this section, we will describe in more detail the building blocks that enabled us to construct the EmStar suite of tools and services. EmStar systems encapsulate logically separable modules within individual processes, and enable communication among these modules through message passing via device files. This structure provides for fault isolation and independence of implementation among services and applications.

In principle, EmStar does not specify anything about the implementation of its modules, apart from the POSIX system call interface required to access device files. For example, most EmStar device interfaces can be used interactively from the shell, and EmStar servers could be implemented in any language that supports the system call interface.

In practice, there is much to be gained from using and creating standard libraries. In the case of EmStar we have implemented these libraries in C, and we have adopted the GLib event framework to manage `select()` and to support timers. Using the event framework we encapsulate complex protocol mechanisms in libraries, and integrate them without explicit coordination. The decision to use C, GLib, and the POSIX interface was designed to minimize the effort required to integrate EmStar with arbitrary languages, implementation styles, and legacy codebases.

We will now describe some key building blocks in more detail: the EmStar IPC mechanisms and associated libraries. We will explain them in terms of what they do, how they work, and how they are used.

3.1 FUSD

FUSD, the Framework for User-Space Devices, is essentially a microkernel extension to Linux. FUSD allows device-file callbacks to be proxied into user-space and implemented by user-space programs instead of kernel code. Though implemented in userspace, FUSD drivers can create device files that are semantically indistinguishable from kernel-implemented `/dev` files, from the point of view of the processes that use them. FUSD follows in the tradition of microkernel operating systems that implement POSIX interfaces, such as QNX [29] and GNU HURD [25].

As we will describe in later sections, this capability is used by EmStar modules for both communication with other modules and with users. Of course, many other IPC methods exist in Linux, including sockets, message queues, and named pipes. We have found a number of compelling advantages in using user-space device drivers for IPC among EmStar processes. For example, system call return values come from the EmStar processes themselves, not the kernel; a successful `write()` guarantees that the data has reached the application. Traditional IPC has much weaker semantics, where a successful `write()` means only that the data has been accepted into a kernel buffer, not that it has been read or acknowledged by an application. FUSD-based IPC obviates the need for explicit application-level acknowledgment schemes built on top of sockets or named pipes.

FUSD-driven devices are a convenient way for applications to transport data, expose state, or be configured in a convenient, browseable, named hierarchy—just as the kernel itself uses the `/proc` filesystem. These devices can respond to system calls using custom semantics. For example, a read from a packet-interface device (Section 3.2.2) will always begin at a packet boundary. The customization

of system call semantics is a particularly powerful feature, allowing surprisingly expressive APIs to be constructed. We will explore this feature further in Section 3.2.

3.1.1 FUSD Implementation

The proxying of kernel system calls is implemented using a combination of a kernel module and cooperating user-space library. The kernel module implements a device, `/dev/fusd`, which serves as a control channel between the two. When a user-space driver calls `fusd_register()`, it uses this channel to tell the FUSD kernel module the name of the device being registered. The FUSD kernel module, in turn, registers that device with the kernel proper using `devfs`, the Linux device filesystem. `Devfs` and the kernel do not know anything unusual is happening; it appears from their point of view that the registered devices are simply being implemented by the FUSD module.

FUSD drivers are conceptually similar to kernel drivers: a set of callback functions called in response to system calls made on file descriptors by user programs. In addition to the device name, `fusd_register()` accepts a structure full of pointers to callback functions, used in response to client system calls—for example, when another process tries to open, close, read from, or write to the driver's device. The callback functions are generally written to conform to the standard definitions of POSIX system call behavior. In many ways, the user-space FUSD callback functions are identical to their kernel counterparts.

When a client executes a system call on a FUSD-managed device (e.g., `open()` or `read()`), the kernel activates a callback in the FUSD kernel module. The module blocks the calling process, marshals the arguments of the system call, and sends a message to the user-space driver managing the target device. In user-space, the library half of FUSD unmarshals the message and calls the user-space callback that the FUSD driver passed to `fusd_register()`. When that user-space callback returns a value, the process happens in reverse: the return value and its side-effects are marshaled by the library and sent to the kernel. The FUSD kernel module unmarshals the message, matches it with the corresponding outstanding request, and completes the system call. The calling process is completely unaware of this trickery; it simply enters the kernel once, blocks, unblocks, and returns from the system call—just as it would for a system call to a kernel-managed device.

One of the primary design goals of FUSD is *stability*. A FUSD driver cannot corrupt or crash any other part of the system, either due to error or malice. Of course, a buggy driver may corrupt itself (e.g., due to a buffer overrun). However, strict error checking is implemented at the user/kernel boundary, which prevents drivers from corrupting the kernel or any other user-space process—including other FUSD drivers, and even the processes using the devices provided by the errant driver.

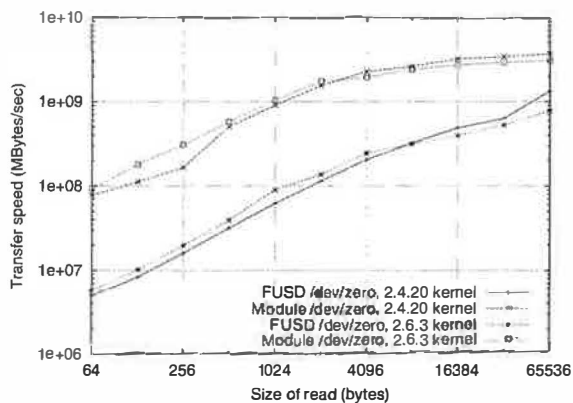


Figure 3: Throughput comparison of FUSD and in-kernel implementations of `/dev/zero`. The test timed a read of 1GB of data from each test device on a 2.8 GHz Xeon, for both 2.4 and 2.6 kernels. We tested `read()` sizes ranging from 64 bytes to 64 Kbytes. Larger read sizes are higher throughput because the cost of a system call is amortized over more data.

3.1.2 FUSD Performance

While FUSD has many advantages, the performance of drivers written using FUSD suffers relative to an in-kernel implementation. To quantify the costs of FUSD, we compared the performance of FUSD and in-kernel implementations of the `/dev/zero` device in Linux. To implement `/dev/zero` using FUSD, we implemented a server with a `read()` handler that returned a zeroed buffer of the requested length. The in-kernel implementation implemented the same `read()` handler directly in the kernel.

Figure 3 shows the results of our experiment, running on a 2.8 GHz Xeon. The figure shows that for small reads, FUSD is about 17x slower than an in-kernel implementation, while for long reads, FUSD is only about 3x slower. This reduction in performance is a combination of two independent sources of overhead.

The first source of overhead is the additional system call overhead and scheduling latency incurred when FUSD proxies the client's system call out to the user-space server. For each `read()` call by a client process, the user-space server first be scheduled, and then must itself call `read()` once to retrieve the marshalled system call, and must call `writenv()` once to return the response with the filled data buffer. This additional per-call latency dominates for small data transfers.

The second source of overhead is an additional data copy. Where the native implementation only copies the response data back to the client, FUSD copies the response data twice: once to copy it from the user-space server, and again to copy it back to the client. This cost dominates for large data transfers.

In our experiments, we tested both the 2.6 and 2.4 kernels, and found that 2.6 kernels yielded an improvement for smaller transfer sizes. The 2.6 kernel has a more significant impact when many processes are running in parallel, as shown in the results of our tests of EmStar simulations

in Section 4.1.4. Further performance analysis of specific EmStar FUSD-based interfaces appears in Section 3.3.2.

3.2 Device Patterns

Using FUSD, it is possible to implement character devices with almost arbitrary semantics. FUSD itself does not enforce any restrictions on the semantics of system calls, other than those needed to maintain fault isolation between the client, server, and kernel. While this absence of restriction makes FUSD a very powerful tool, we have found that in practice the interface needs of most applications fall into well-defined classes, which we term *Device Patterns*. Device Patterns factor out the device semantics common to a class of interfaces, while leaving the rest to be customized in the implementation of the service.

The EmStar device patterns are implemented by libraries that hook into the GLib event framework. The libraries encapsulate the detailed interface to FUSD, leaving the service to provide the configuration parameters and callback functions that tailor the semantics of the device to fit the application. For example, while the Status Device library defines the mechanism of handling each `read()`, it calls back to the application to represent its current "status" as data.

Relative to other approaches such as log files and status files, a key property of EmStar device patterns is their active nature. For example, the Logging Device pattern creates a device that appears to be a regular log file, but always contains only the most recent log messages, followed by a stream of new messages as they arrive. The Status Device pattern appears to be a file that always contains the most recent state of the service providing it. However, most status devices also support `poll()`-based notification of changes to the state.

The following sections will describe the Device Patterns defined within EmStar. Most of these patterns were discovered during the development of services that needed them and later factored out into libraries. In some cases, several similar instances were discovered, and the various features amalgamated into a single pattern.

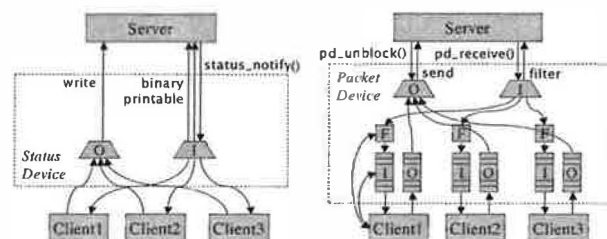


Figure 4: Block diagram of the (a) Status and (b) Packet Device patterns. In the Packet Device diagram, the "F" boxes are client-configurable filters, and the curved arrows from Client1 represent `ioctl()` based configuration of queue lengths and message filtering. Trapezoid boxes represent multiplexing of clients.

3.2.1 Status Device

The Status Device pattern provides a device that reports the current state of a module. The exact semantics of “state” and its representation in both human-readable and binary forms are determined by the service. Status Devices are used for many purposes, from the output of a neighbor discovery service to the current configuration and packet transfer statistics for a radio link. Because they are so easy to add, Status Devices are often the most convenient way to instrument a program for debugging purposes, such as the output of the Neighbors service and the packet reception statistics for links.

Status Devices support both human-readable and binary representations through two independent callbacks implemented by the service. Since the devices default to ASCII mode on `open()`, programs such as `cat` will read a human-readable representation. Alternatively, a client can put the device into binary mode using a special `ioctl()` call, after which the device will produce output formatted in service-specific structs. For programmatic use, binary mode is preferable for both convenience and compactness.

Status Devices support traditional read-until-EOF semantics. That is, a status report can be any size, and its end is indicated by a zero-length read. But, in a slight break from traditional POSIX semantics, a client can keep a Status Device open after EOF and use `poll()` to receive notification when the status changes. When the service triggers notification, each client will see its device become readable and may then read a new status report.

This process highlights a key property of the status device: while every new report is guaranteed to be the current state, a client is not guaranteed to see every intermediate state transition. The corollary to this is that if no clients care about the state, no work is done to compute it. Applications that desire queue semantics should use the Packet Device pattern (described in Section 3.2.2).

Like many EmStar device patterns, the Status Device supports multiple concurrent clients. Intended to support one-to-many status reporting, this feature has the interesting side effect of increasing system transparency. A new client that opens the device for debugging or monitoring purposes will observe the same sequence of state changes as any other client, effectively snooping on the “traffic” from that service to its clients. The ability to do this interactively is a powerful development and troubleshooting tool.

A Status Device can implement an optional `write()` handler, which can be used to configure client-specific state such as options or filters. For example, a routing protocol that maintained multiple routing trees might expose its routing tables as a status device that was client-configurable to select only one of the trees.

3.2.2 Packet Device

The Packet Device pattern provides a read/write device that provides a queued multi-client packet interface. This pattern is generally intended for packet data, such as the interface to a radio, a fragmentation service, or a routing service, but it is also convenient for many other interfaces where queue semantics are desired.

Reads and writes to a Packet Device must transfer a complete packet in each system call. If `read()` is not supplied with a large enough buffer to contain the packet, the packet will be truncated. A Packet Device may be used in either a blocking or `poll()`-driven mode. In `poll()`, readable means there is at least one packet in its input queue, and writable means that a previously filled queue has dropped below half full.

Packet Device supports per-client input and output queues with client-configurable lengths. When at least one client's output queue contains data, the Packet Device processes the client queues serially in round-robin order, and presents the server with one packet at a time. This supports the common case of servers that are controlling access to a rate-limited serial channel.

To deliver a packet to clients, the server must call into the Packet Device library. Packets can be delivered to individual clients, but the common case is to deliver the packet to all clients, subject to a client-specified filter. This method enhances the transparency of the system by enabling a “promiscuous” client to see all traffic passing through the device.

3.2.3 Command Device

The Command Device pattern provides an interface similar to the writable entries in the Linux `/proc` filesystem, which enable user processes to modify configurations and trigger actions. In response to a `write()`, the provider of the device processes and executes the command, and indicates any problem with the command by returning an error code. Command Device does not support any form of delayed or asynchronous return to the client.

While Command Devices can accept arbitrary binary data, they typically parse a simple ASCII command format. Using ASCII enables interactivity from the shell and often makes client code more readable. Using a binary structure might be slightly more efficient, but performance is not a concern for low-rate configuration changes.

The Command Device pattern also includes a `read()` handler, which is typically used to report “usage” information. Thus, an interactive user can get a command summary using `cat` and then issue the command using `echo`. Alternatively, the Command Device may report state information in response to a read. This behavior would be more in keeping with the style used in the `/proc` filesystem, and is explicitly implemented in a specialization of Command Device called the Options Device pattern.

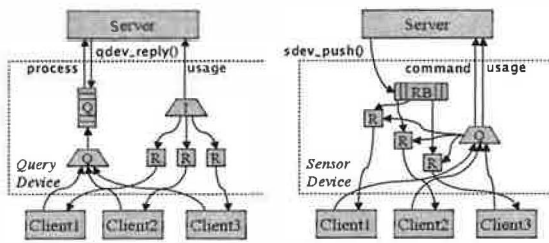


Figure 5: Block diagram of the (a) Query and (b) Sensor Device patterns. In the Query Device, queries from the clients are queued and “process” is called serially. The “R” boxes represent a buffer per client to hold the response to the last query from that client. In the Sensor Device, the server submits new samples by calling `sdev_push()`. These are stored in the ring buffer (RB), and streamed to clients with relevant requests. The “R” boxes represent each client’s pending request.

3.2.4 Query Device

The Device Patterns we have covered up to now provide useful semantics, but none of them really provides the semantics of RPC. To address this, the Query Device pattern implements a transactional, request/response semantics. To execute a transaction, a client first opens the device and writes the request data. Then, the client uses `poll()` to wait for the file to become readable, and reads back the response in the same way as reading a Status Device. For those services that provide human-readable interfaces, we use a universal client called `echocat` that performs these steps and reports the output.

It is interesting to note that the Query Device was not one of the first device types implemented; rather, most configuration interfaces in EmStar have been implemented by separate Status and Command devices. In practice, any given configurable service will have many clients that need to be apprised of its current configuration, independent of whether they need to change the configuration. This is exacerbated by the high level of dynamics in sensor network applications. Furthermore, to build more robust systems we often use soft-state to store configurations. The current configuration is periodically read and then modified if necessary. The asynchronous Command/Status approach achieves these objectives while addressing a wide range of potential faults.

To the service implementing a Query Device, this pattern offers a simple, transaction-oriented interface. The service defines a callback to handle new transactions. Queries from the client are queued and are passed serially to the transaction processing callback, similar to the way the output queues are handled in a Packet Device. If the transaction is not complete when the callback returns, it can be completed asynchronously. At the time of completion, a response is reported to the device library, which it then makes available to the client. The service may also optionally provide a callback to provide usage information, in the event that the client reads the device before any query has been sub-

mitted.

Clients of a Query Device are normally serviced in round-robin order. However, some applications need to allow a client to “lock” the device and perform several back-to-back transactions. The service may choose to give a current client the “lock”, with an optional timeout. The lock will be broken if the timeout expires, or if the client with the lock closes its file descriptor.

3.3 Domain-Specific Interfaces

In Section 3.2 we described several device patterns, generally useful primitives that can be applied to a wide variety of purposes. In this section, we will describe a few examples of more domain-specific interfaces, that are composed from device patterns, but are designed to support the implementation of specific types of services.

3.3.1 Data Link Interface

The Data Link interface is a specification of a standard interface for network stack modules. The Data Link interface is composed of three device files: `data`, `command`, and `status`. These three interfaces appear together in a directory named for the specific stack module.

The data device is a Packet Device interface that is used to exchange packets with the network. All packets transmitted on this interface begin with a standard link header that specifies common fields. This link header masks certain cosmetic differences in the actual over-the-air headers used by different MAC layers, such as the Berkeley MAC [17] and SMAC [22] layers supported on Mica Motes.

The command and status devices provide asynchronous access to the configuration of a stack module. The status device reports the current configuration of the module (such as its channel, sleep state, link address, etc.) as well as the latest packet transfer and error statistics. The command device is used to issue configuration commands, for example to set the channel, sleep state, etc. The set of valid commands and the set of values reported in status varies with the underlying capabilities of the hardware. However, the binary format of the status output is standard across all modules (currently, the union of all features).

Several “link drivers” have been implemented in EmStar, to provide interfaces to radio link hardware including 802.11, and several flavors of the Mica Mote. The 802.11 driver overlays the socket interface, sending and receiving packets through the Linux network stack. Two versions of the Mote driver exist, one that supports the standard Berkeley MAC and one that supports SMAC. Because all of these drivers conform to the link interface spec, some applications can work more or less transparently over different physical radio hardware. In the event that an application needs information about the radio layer (e.g. the nominal link capacity), that information is available from the status device.

In addition to providing support for multiple underlying

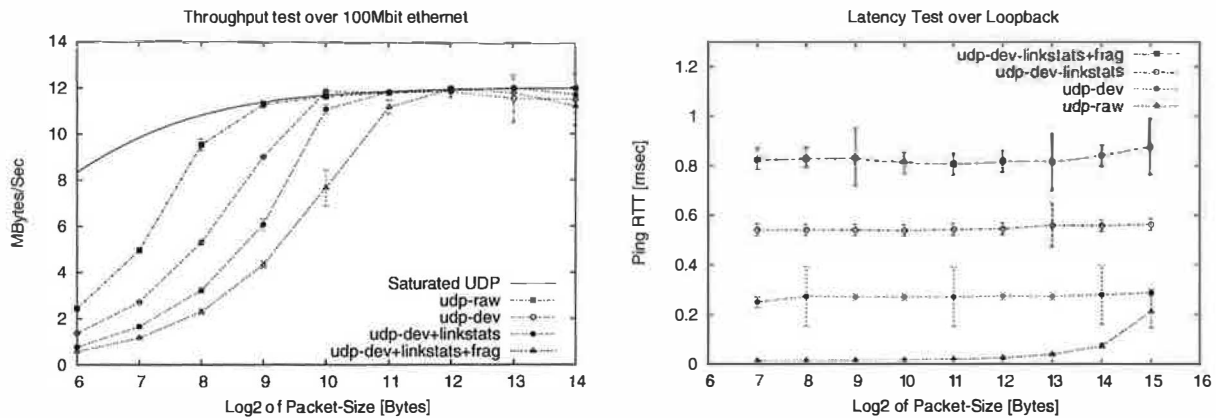


Figure 6: Measurements of the EmStar stack on a 700 MHz Pentium III running the 2.4.20 kernel. The throughput graph shows the performance of a single process sending at maximum rate over a 100Mbit Ethernet, as a function of packet length, through different EmStar stacks. The solid curve represents link saturation, while the other curves compare the performance of sending directly to a socket with that of sending through additional layers. The error bars are 95% confidence intervals. The latency graph shows the average round-trip delay of a ping message over the loopback interface, as a function of packet length, through different EmStar stacks. Both graphs show that performance is dominated by per-packet overhead rather than data transfer, consistent with previous results about FUSD.

radio types, the standard Data Link interface enables a variety of useful “pass-through” stack modules and routing modules. Two standard modules in EmStar network stacks are LinkStats and Fragmentation. Both of these sit between a client and an underlying radio driver module, transparently to the client. In addition to passing data through, they proxy and modify status information, for example updating the MTU specification.

3.3.2 Cost Analysis of the Data Link Interface

Our discussion up to this point has yet to address the cost of this architecture. In order to quantify some of these costs, we performed a series of experiments, the results of which are shown in Figure 6. We found that while our architecture introduces a measurable increase in latency and decrease in throughput relative to a highly integrated and optimized solution, these costs have a negligible impact when applied to a low bandwidth communications channel. This is an important case, since EmStar is intended for WSN applications which typically are designed to have a high ratio of CPU to communication.

To assess the costs of EmStar, we measured the costs incurred by layering additional modules over an EmStar link device. The `udp-raw` curves represent a non-EmStar benchmark, in which we used a UDP socket directly. The `udp-dev` curves represent a minimal EmStar configuration, in which we used the EmStar UDP Link device. For a two-layer stack, we added the EmStar LinkStats module, represented by the `+linkstats` curves. For a three-layer stack, we added a Fragmentation module over LinkStats, shown by the `+frag` curves.

Our first experiment characterized the cost of EmStar in terms of throughput. In Figure 6(a), our test application sent UDP packets as quickly as possible over a 100Mbit

Ethernet channel. We ran this application over our four configurations, comparing direct sends to a socket with three EmStar configurations. For each configuration, the time required to send 1000 packets was measured, and the results of 10 such trials were averaged. The graph shows that per-packet overhead prevents the application from saturating the link until larger packet sizes sufficiently amortize the per-packet costs. Per-packet costs include scheduling latency and system call overhead, while message-passing across the user-kernel boundary results in additional per-byte costs.

Our second experiment characterized the cost of EmStar in terms of latency. In Figure 6(b), our test application sent UDP “ping” packets over the loopback interface to a ping replier on the same machine. We measured the round-trip times for 1000 packets and averaged them to estimate the latency for our four configurations. Since the latency over loopback is negligible (shown in the “`udp-raw`” curve), all of the measured latency represents EmStar overhead. In each case, a ping round trip traverses the stack four times, thus is approximately 4x the latency of a single traversal. The data show that crossing an EmStar interface costs about 66 microseconds on this architecture, without a strong dependence on the length of the message being passed.

While these experiments show definite costs to the EmStar architecture, these costs are less critical for WSN applications where communications channels have lower bandwidths and higher latency relative to the rate of local processing. For example, many of our applications use a Mote as a radio interface, which has a maximum bandwidth of about 19.2Kbit/sec and incurs a latency of 125 milliseconds to transmit a 200 byte packet over serial to the Mote and then over the channel. Given this type of interface, the

additional latency and bandwidth costs of EmStar are negligible.

3.3.3 Sensor Device

Two of the applications that drove the development of EmStar centered around acquisition and processing of audio data. One application, a ranging and localization system [15], extracts and processes audio clips from a specific time in the past. The other, a continuous frog call detection and localization system [8], receives data in a continuous stream. Both applications needed to be able to correlate time series data captured on a distributed set of nodes, thus timing relationships among the nodes needed to be maintained.

The Sensor Device interface encapsulates a ring buffer that stores a history of sampled data, and integrates with the EmStar Time Synch service to enable clients to relate local sensor data to sensor data from other nodes. A client of the sensor device can open the device and issue a request for a range of samples. When the sample data is captured, the client is notified and the data is streamed back to the client as it continues to arrive.

Keeping a history of recent sensor data and being able to relate the sample timing across the network is critical to many sensor network applications. By retaining a history of sampled data, it is much easier to implement applications where an event detected on one node triggers further investigation and sensing at other nodes. Without local buffering, the variance in multi-hop communications times makes it difficult to abstract the triggered application from the communications stack.

3.4 EmStar Events and Client APIs

One of the benefits of the EmStar design is that services and applications are separate processes and communicate through POSIX system calls. As such, EmStar clients and applications can be implemented in a wide variety of languages and styles. However, a large part of the convenience of EmStar as a development environment comes from a set of helper libraries that improve the elegance and simplicity of building robust applications.

In Section 3.2 we noted that an important part of device patterns is the library that implements them on the service side. Most device patterns also include a client-side “API” library, that provides basic utility functions, GLib compatible notification interfaces, and a *crashproofing* feature intended to prevent cascading failures.

Crashproofing is intended to prevent the failure of a lower-level service from causing exceptions in clients that would lead them to abort. It achieves this by encapsulating the mechanism required to open and configure the device, and automatically triggering that mechanism to re-open the device whenever it closes unexpectedly.

A client’s use of crashproof devices is completely transparent. The client constructs a structure specifying the device name, a handler callback, and the client configura-

tion, including desired queue lengths, filters, etc. Then, the client calls a constructor function that opens and configures the device, and starts watching it. In the event of a crash and reopen, the information originally provided by the client will be used to reconfigure the new descriptor. Crashproof client libraries are supplied for both Packet and Status devices.

4 Examples

The last section enumerated a number of building blocks that are the foundation for the EmStar environment. In this Section, we will describe how we have used them to construct several key EmStar tools and services.

4.1 EmSim and EmCee

EmSim and EmCee are tools designed to simulate unmodified EmStar systems at varying points on the continuum from simulation to deployment. EmSim is a pure simulation environment, in which many virtual nodes are run in parallel, interacting with a simulated environment and radio channel. EmCee is a slightly modified version of EmSim that provides an interface to real low-power radios in place of a simulated channel.

EmSim itself is made up of modules. The main EmSim module maintains a central repository for node information, initially sourced from a configuration file, and exposed as a Status Device. EmSim then launches other modules that are responsible for implementing the simulated “world model” based on the node configuration. After the world is in place, EmSim begins the simulation, starting up and shutting down virtual nodes at the appropriate times.

4.1.1 Running Virtual Nodes

The uniform use of the `/dev` filesystem for all of our I/O and IPC leads to a very elegant mechanism for transparency between simulation, various levels of reality, and real deployments. The mechanism relies on name mangling to cause all references to `/dev/*` to be redirected deeper into the hierarchy, to `/dev/sim/groupX/nodeY/*`. This is achieved through two simple conventions.

First, all EmStar modules must include the call to `misc_init()` early in their `main()` function. This function checks for certain environment variables to determine whether the module is running in “simulation mode”, and what its group and node IDs are. The second convention is to wrap every instance of a device file name with `sim_path()`. This macro will perform name-mangling based on the information discovered in `misc_init()`. For simplicity, we typically include the `sim_path()` wrapper at the definition of device names in interface header files.

This approach enables easy and transparent simulation of many nodes on the same machine. This is not the case for many other network software implementations. Whenever the system being developed relies on mechanisms inside the kernel that can’t readily be partitioned into virtual machines, it will be difficult to implement a transparent

simulation.

For example, ad-hoc routing code that directly configures the network interfaces and kernel routing table is very difficult to simulate transparently. While a simulation environment such as *ns-2* [27] does attempt to run much of the same algorithmic code as the real system, it does so in a very intrusive, *#ifdef*-heavy way. This makes it cumbersome to keep the live system in sync with the *ns-2* version.

In contrast, EmStar modules don't even need to be recompiled to switch from simulation to reality, and the EmStar device hierarchy provides transparency into the workings of each simulated EmStar node. However, this flexibility comes at a cost in performance. An ad-hoc routing algorithm that dragged every packet to user-space would likely suffer poorer performance.

4.1.2 Simulated World Models

The capability to transparently redirect EmStar IPC channels enables us to provide a world for the simulated nodes to see, and in some cases, affect. There are many examples of network simulation environments in the networking community, some of which support radio channel modeling [27][28]. In addition, the robotics community has devoted much effort to creating world models [12]. For sensor networks, the robotic simulations are often more appropriate, because they are designed to model a system sensing the environment, and intended to test and debug control systems and behaviors that must be reactive and resilient.

The existence of EmStar device patterns simplifies the construction of simulated devices, because all of the complexity of the interface behavior can be reused. Even more important, by using the same libraries, the chances of subtle behavior differences are reduced. Typically, a "simulation module" reads the node configuration from EmSim's Status Device and then exposes perhaps hundreds of devices, one for each node. Requests to each exposed device are processed according to a simulation of the effects of the environment, or in some cases in accordance with traces of real data.

The notification channel in EmStar status devices enables EmSim to easily support configurations changes during a simulation. Updates to the central node configuration—such as changes in the position of nodes—trigger notification in the simulation modules. The modules can then read the new configuration and update their models appropriately. In addition, we can close the loop by creating a simulation module that provides an actuation interface—for example enabling the node to move itself. In response to a request to move, this module could issue a command to EmSim to update that node's position and notify all clients.

4.1.3 Using Real Channels in the Lab

EmCee is a variant of EmSim that integrates a set of virtual nodes to a set of real radio interfaces, positioned out in the world. We have two EmCee-compatible testbeds:

the ceiling array and the portable array. The ceiling array is composed of 55 Crossbow Mica1 Motes, permanently attached to the ceiling of our lab on a 4 foot grid. Serial cabling runs back to two 32-port ethernet to serial multiplexers. The portable array is composed of 16 Crossbow Mica2 Motes and a 16-port serial multiplexer, that can be taken out to the field [6].

The serial multiplexers are configured so that their serial ports appear to be normal serial devices on a Linux server (or laptop in the portable case). To support EmCee, the HostMote and MoteNIC services support an "EmCee mode" where they open a set of serial ports specified in a config file and expose their devices within the appropriate virtual node spaces.

Thus, the difference between EmSim and EmCee is minimal. Where EmSim would start up a radio channel simulator to provide virtual radio link devices, EmCee starts up the MoteNIC service in "EmCee mode", which creates real radio link devices that map to multiplexer serial ports and thus to real Motes.

Our experience with EmCee has shown it is well worth the infrastructure investment. Users have consistently observed that using real radios is substantially different from our best efforts at creating a modeled radio channel [2][6]. Even channels driven by empirical data captured using the ceiling array don't seem to adequately capture the real dynamics. Although testing with EmCee is still not the same as a real deployment, the reduction in effort relative to a deployment far outweighs the reduction in reality for a large part of the development and testing process.

4.1.4 Performance of EmSim/EmCee

Currently, an important limitation of our simulator is that it can only run in real-time, using real timers and interrupts from the underlying operating system. In contrast, a discrete-event simulator such as *ns-2* runs in its own virtual time, and therefore can run for as long as necessary to complete the simulation without affecting the results. Discrete-event simulations can also be made completely deterministic, allowing the developer to more easily reproduce an intermittent bug.

The real-time nature of EmSim/EmCee makes performance an important consideration. With perfect efficiency, the simulator platform would need the aggregate computational power of all simulated nodes. In reality, extra headroom is needed for nonlinear costs of running many processes on a single computer.

To test the actual efficiency, we ran test simulations on two SMP-enabled servers. One had 4 700MHz Pentium-III processors, running Linux kernel 2.4.20. The other had 2 2.8GHz Xeon processors, with hyperthreading disabled, running Linux 2.6.3. We tested both kernels because Linux 2.6 has a "O(1) scheduler"—i.e., the 2.6 scheduler performs constant work per context switch regardless of run-queue size. 2.6 kernels also have much finer-grained locking, thus better kernel parallelism. The FUSD kernel

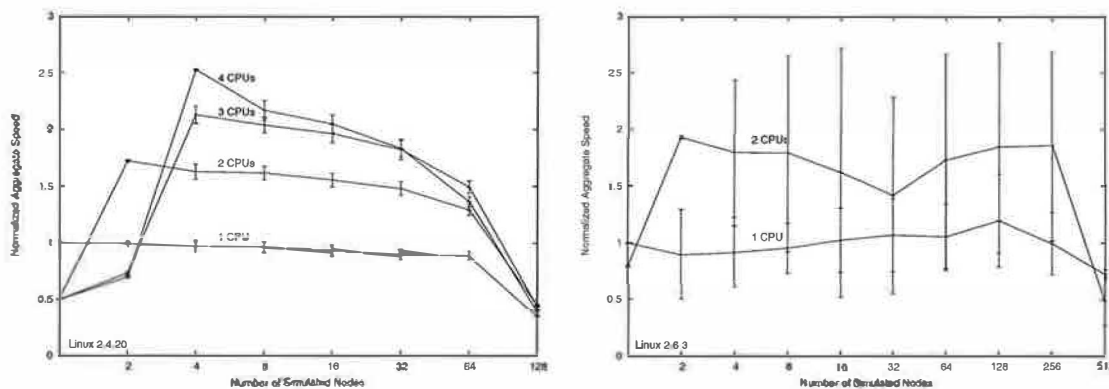


Figure 7: Performance of a simple EmSim simulation, varying the number of nodes simulated, and the number of CPUs available on the simulator platform. Linux kernels 2.4.20 (*left*) and 2.6.3 (*right*) were tested. Each “node” is two processes that continuously exchange data via a EmStar Status Interface. We plot the aggregate transfer rate summed across all simulated nodes. Results are normalized so that $y = 1$ corresponds to the speed achieved by a single-node simulation (2 processes) running on a single CPU.

module also has fine-grained locking.

In our initial testing, the default Linux scheduler was used; no explicit assignment of processes to CPUs was made. Each “node” consisted of two processes that exchanged data at maximum possible rate via a EmStar Status Device. The results are in Figure 7.

We draw several conclusions from the data. First, the Linux 2.6 scheduler does seem to be a win. Even with differences in CPU speed factored out, it supported much larger simulations than the 2.4 scheduler (512 vs. 128 nodes). In addition, it supported better parallelism: Linux 2.6 with 2 CPUs had, on average, 1.7 times more throughput than for a single CPU, compared to 1.5 times for Linux 2.4. However, Linux 2.6 simulations suffered much higher jitter, i.e. differences in performance from node to node. The cause of this unfairness is still under investigation.

The data also emphasize the high cost of FUSD inter-process communication across processes not running on the same CPU. This can be seen in that a single-node (2-process) simulation ran on a single-CPU platform at nearly at nearly *twice the speed* as on 2-, 3- or 4-CPU platform. The Linux scheduler, by default, places the Status Device Client and Server processes on separate CPUs if available. In applications that have a very high communication-to-computation ratio, as in our test workload, the overhead of extra CPUs is a much higher cost than the benefit of extra cycles. However, many EmStar applications (and WSN applications in general) strive to do as much computation as possible per unit of communication, making these limitations of SMP model a virtual non-issue in “real” simulations.

4.2 EmRun

EmRun starts up, maintains, and shuts down an EmStar system according to the policy specified in a config file. There are three key points in its design: process respawn, in-memory logging, and fast startup, graceful shutdown.

Respawn Process respawn is neither new, nor difficult to achieve, but it is very important to an EmStar system. It is difficult to track down every bug, especially ones that occur very infrequently, such as a floating-point error processing an unusual set of data. Nonetheless, in a deployment, even infrequent crashes are still a problem. Often, process respawn is sufficient to work around the problem; eventually, the system will recover. EmStar’s process respawn is unique because it happens in the context of “Crashproofed” interfaces (Section 3.4). When an EmStar process crashes and restarts, Crashproofing prevents a ripple effect, and the system operates correctly when the process is respawned.

In-Memory Logs EmRun saves each process’ output to in-memory log rings that are available interactively from the `/dev/emlog/*` hierarchy. These illustrate the power of FUSD devices relative to traditional logfiles. Unlike rotating logs, EmStar log rings never need to be switched, never grow beyond a maximum size, and always contain only recent data.

Fast Startup EmRun’s fast startup and graceful shutdown is critical for a system that needs to duty cycle to conserve energy. The implementation depends on a control channel that EmStar services establish back to EmRun when they start up. EmStar services notify EmRun when their initialization is complete, signaling that they are now ready to respond to requests. The `emrun_init()` library function, called by the service, communicates with EmRun by writing a message to `/dev/emrun/.int/control`. EmRun then launches other processes waiting for that service, based on a configured dependency graph.

This feedback enables EmRun to start independent processes with maximal parallelism, and to wait *exactly* as long as it needs to wait before starting dependent processes. This scheme is far superior to the naive approach of waiting between daemon starts for pre-determined times, i.e., the ubiquitous “sleep 2” statements found in *NIX boot scripts.

Various factors can make startup times difficult to predict and high in variance, such as flash filesystem garbage collection. On each boot, a static sleep value will either be too long, causing slow startup, or too short, causing services to fail when their prerequisites are not yet available.

Graceful Shutdown The control channel is also critical to supporting graceful shutdown. EmRun can send a message through that channel, requesting that the service shut down, saving state if needed. EmRun then waits for SIGCHLD to indicate that the service has terminated. If the process is unresponsive, it will be killed by a signal.

An interesting property of the EmRun control channel is one that differentiates FUSD from other approaches. When proxying system calls to a service, FUSD includes the PID, UID, and GID of the client along with the marshalled system call. This means that EmRun can implicitly match up the client connections on the control channel to the child processes it has spawned, and reject connections from non-child processes. This property is not yet used much in EmStar but it provides an interesting vector for customizing device behavior.

4.3 Time-Synchronized Sampling in EmStar

Several of the driving applications for EmStar have involved distributed processing of high-rate audio: audible acoustic ranging, acoustic beamforming, and animal call detection are a few of the applications. We used earlier versions of EmStar to tackle a few of these problems [10][15][8]. Referring back to the animal call localization application of Figure 1, we see how the “syncd” and “audiod” services collaborate so that “collab_detect” can correlate events detected on nodes across the network. In this section, we will describe these services in more detail.

TimeSync Between Nodes The TimeSync service uses Reference Broadcast Synchronization (RBS) [3] to compute relationships among the CPU clocks on nodes in a given broadcast domain. This technique correlates the arrival times of broadcast packets at different nodes and uses linear regression to estimate conversion parameters among clocks that receive broadcasts in common. We chose RBS because techniques based on measuring send times, such as TPSN [14], are not generally applicable without support at the MAC layer. Requiring this support would rule out many possible radios, including 802.11 cards.

A key insight in RBS is that it is better to enable conversion than to attempt to train a clock to follow some remote “master” clock. Training a clock has many negative repercussions for the design of a sampling system caused by clock discontinuities and distortions.

Thus, TimeSync is really a “time conversion” service. The output of the regression is reported through the /dev/sync/params/ticker device, in a complete listing of all known pairwise conversions. Clients of TimeSync read this device to get the latest conversion parameters, then convert times from one timebase to another. The code for reading

the device and converting among clocks is implemented in a library.

TimeSync within a Node Many systems have more than one clock. For example, a Stargate board, with an attached Mote and an audio card has three independent clocks. Thus to compare audio time series from two independent nodes, an index in a time series must be converted first to local CPU time, then to remote CPU time, and finally to a remote audio sample index.

The TimeSync service provides an interface for other services to supply pair-wise observations to it, i.e. a CPU timestamp and a clock-X timestamp. This interface uses a Directory device to enable clients to create a new clock, and associate it with a numeric identifier. The client then writes periodic observations of that clock to the timesync command device /dev/sync/params/command. The observations are fit using linear regression to compute a relationship between the two local clocks.

The Audio Server The Audio service provides a Sensor Device output. It defines a “sample clock”, which is the index of samples in a stream, and submits observations relating the sample clock to the CPU time to TimeSync.

A client of the Audio service can extract a sequence of data from a specific time period by first using TimeSync to convert the begin and end times to sample indices and then placing a request to the Audio service for that sample range. Conversely, a feature detected in the streaming output at a particular sample offset can be converted to a CPU time. These clock relations can also be used to compute and correct the skew in sample rates between devices, which can otherwise cause significant problems.

Generating the synch observations requires minor changes to the audio driver in the kernel. We have made patches for two audio drivers: the iPAQ built-in audio driver and the Crystal cs4281. In both cases, incoming DMA interrupts are timestamped and retrieved by the Audio service via ioctl(). While this approach makes the system harder to port to new platforms and hardware, it is a better solution for building sensing platforms.

The more common solution, the “synchronized start” feature of many sound cards, has numerous drawbacks. First, it only gives you one data point for the run, where our technique gives you a continuous stream of points to average. Second, it is subject to drift, and since the end is not timestamped there is no way to accurately determine the actual sample rate. Third, it forces the system to coordinate use of the audio hardware, whereas the Audio server runs continuously and allows access by multiple clients.

5 Design Philosophy and Aesthetics

In this section, we will describe some of the ideas behind the choices we made in the design of EmStar. These choices were motivated by the issues faced by WSNs, which have much in common with traditional distributed systems.

5.1 No Local/Remote Transparency

One of the disadvantages of FUSD relative to sockets is that connections to FUSD services are always local, whereas sockets provide transparency between local and remote connections. Nonetheless, we elected to base EmStar on FUSD because we felt that the advantages outweighed the disadvantages.

The primary reason for giving up remote transparency in EmStar is that remote access is rarely transparent in WSNs. Communications links in WSNs are characterized by high or variable latency, varying link quality, evolving topologies, and generally low bandwidth. In addition, the energy cost of communication in WSNs motivates innovative protocols that minimize communications, make use of broadcast channels, tolerate high latency, and make tradeoffs explicit to the system designer. Remote communication in WSNs is demonstrably different than local communication, and very little is achieved by masking that fact.

In abandoning remote transparency, the client gains the benefit of knowing that each synchronous call will be received and processed by the server with low latency. While an improperly implemented server can introduce delays, there is never a need to worry that the network might introduce unexpected delay. Requests that are known to be time consuming can be explicitly implemented so that the results are returned asynchronously via notification (e.g. Query Device).

5.2 Intra-Node Fault Tolerance

Tolerance of node and communications failures is important to the design of all distributed systems. In WSNs, node robustness takes on an even greater importance. First, the cost of replacing or repairing embedded nodes can be much higher, especially when network access to the node is unreliable or a physical journey is required—in extreme cases, nodes may be physically irretrievable. Second, many scientific applications of WSNs intend to discover new properties of their environment, which may expose the system to new inputs and exercise new bugs.

We address fault tolerance within a node in several ways: EmRun respawn, “crashproofing”, soft-state refresh, and transactional interface design. We discussed EmRun respawn and crashproofing in Sections 4.2 and 3.4, as a means of keeping the EmStar services running and preventing cascading failures when an underlying service fails.

While soft-state and transactional design are standard techniques in distributed systems, in EmStar we apply these techniques to IPC as well. Status devices are typically used in a soft-state mode. Rather than reporting more economical “diffs”, every status update reports the complete current state, leaving the client to decide how to respond based on its own state. To limit the damage caused by a missing notification signal, clients periodically request a refresh in the absence of notification. When the aggregate update rate is low it is usually easy to make the case for trading efficiency

for robustness and simplicity.

Similar considerations hold in the reverse direction. Clients that push state to a service typically use transactional semantics with a soft-state refresh. Rather than allowing the client and server to get out of synch (e.g. in the event of a server restart), the client periodically resubmits its *complete* state to the service, enabling the service to make appropriate corrections if there is a discrepancy. Where the state in question is very large, there may be reason to implement a more complex scheme, but for small amounts of state, simplicity and robustness carry the day. While trading off efficiency for robustness may not be the right approach for all applications and hardware platforms, it has worked well for the applications we have built.

5.3 Code Reuse

Code reuse and modularity were major design goals of EmStar. EmStar achieves reusability through disciplined design, driven by factoring useful components from existing implementations. For example, each device pattern was originally implemented as a part of several different services, and then factored out into a unified solution to a class of problems. Table 1 shows a quantitative picture of reuse.

The design of EmStar services has followed the dictum “encapsulate mechanism, not policy”. This approach encourages reuse, and reduces system complexity while maintaining simple interfaces between modules. EmStar implements modules as independent processes rather than as libraries, eliminating a wide variety of unanticipated interactions, thus better controlling complexity as the number of modules increases.

Building Block	Server Uses	Client Uses
Status Device and derivatives	40	22
Command Device	17	N/A
Packet Device	10	5
Data Link Interface	12	32

Table 1: Reuse statistics culled from LXR.

5.4 Reactivity

Reactivity is one of the most interesting characteristics of WSNs. They must react to hard-to-predict changes in their environment in order to operate as designed. Often the tasks themselves require a reaction, for example a distributed control system or a distributed sensing application that triggers other sensing activities. EmStar supports reactivity through notification interfaces in EmStar devices. Most EmStar services and applications are written in an event-driven style that lends itself to reactive design.

5.5 High Visibility

While the decision to stress visibility in the EmStar design was partly motivated by aesthetics, it has paid off handsomely in overall ease of use, development, and debugging. The ability to browse the IPC interfaces in the shell, to see human-readable outputs of internal state, and in many

cases to manually trigger actions makes for very convenient development of a system that could otherwise be quite cumbersome. Tools like EmView also benefit greatly from stack transparency, because EmView can snoop on traffic travelling in the middle of the stack in real time, without modifying the stack itself.

6 Related Work

In addition to related work we mentioned throughout this paper, in this section we highlight the most related systems.

The closest system to EmStar is TinyOS [17]. TinyOS addresses the same problem space, only geared to the much smaller Mote platform. As such, much TinyOS development effort must focus on reducing memory and CPU usage. By operating with fewer constraints, EmStar can focus on more complex applications and on improving robustness in the face of growing complexity. A key attribute of TinyOS that EmStar lacks is the capacity to perform system-wide compile time optimizations. Because EmStar supports forms of dynamic binding that do not exist in TinyOS, many compile-time optimizations are ruled out.

Click [19] is a modular software system designed to support implementation of routers. While Click is designed for a different application space, there are many similarities, including an emphasis on modularity. A key difference is that like TinyOS, Click leverages language properties and static configuration to perform global optimizations. EmStar instead supports dynamic configuration and provides greater levels of fault isolation between modules.

Player/Stage [12] is a software system designed for robotics that supports “real-code” simulation. Player is based on sockets protocols, which have the advantage of remote transparency but are not browseable.

7 Conclusion and Future Work

We have found EmStar to be a very useful development environment for WSNs. We use EmStar at CENS in several current development efforts, including a 50-node seismic deployment and the ESS microclimate sensing system. We also support other groups using EmStar, including the NIMS [13] robotic ecology project and ISI ILENSE.

Our current platform focus is the Crossbow/Intel Stargate platform, an inexpensive Linux platform based on the XScale processor. Stargates are much easier to customize than other COTS platforms such as iPAQs.

We plan several extensions to EmStar, including: better integration between Motes and Microservers based on a TinyOS “VM”, virtualization of EmSim’s clock to enable “simulation pause” and larger simulations, remote device access over local networks via sockets, and efficient support for high-bandwidth sensor interfaces such as audio, image data, and DSPs using a shared-memory data channel.

Acknowledgements

This work was made possible with support from The Center for Embedded Networked Sensing (CENS) under the NSF

Cooperative Agreement CCR-0120778, and the UC MICRO program (grant 01-031) with matching funds from Intel Corp. Additional support from the DARPA NEST program (the “GALORE” project, grant F33615-01-C-1906).

References

- [1] In *Proc. of the First Intl. Conf. on Embedded Networked Sensor Systems (ACM Sensys)*, Los Angeles, CA, November 2003.
- [2] H. Dubois-Ferriere. Using voronoi clusters to scope floods from multiple sinks. Lecture, CENS Systems Seminar, October 2003.
- [3] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI*, pages 147–163, Boston, MA, December 2002.
- [4] D. Estrin, W. Michener, G. Bonito, and Workshop Participants. Environmental Cyberinfrastructure Needs for Distributed Sensor Networks: A Report From an NSF Workshop. Workshop Report, Scripps Institute of Oceanography, La Jolla, CA, August 2003.
- [5] A. Cerpa et. al. Habitat Monitoring: Application Driver for Wireless Communications Technology. In *2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, San Jose, Costa Rica, April 2001.
- [6] A. Cerpa et. al. SCALE: A tool for Simple Connectivity Assessment in Lossy Environments. *CENS TR 0021*, September 2003.
- [7] A. Mainwaring et. al. Wireless Sensor Networks for Habitat Monitoring. In *WSNA*, number 1, 2002.
- [8] H. Wang et. al. Target Classification and Localization in Habitat Monitoring. In *ICASSP 2003*, Hong Kong, China, April 2003.
- [9] J. Elson et. al. EmStar: An Environment for Developing Wireless Embedded Systems Software. *CENS TR 0009*, March 2003.
- [10] J.C. Chen et. al. Coherent Acoustic Array Processing and Localization on Wireless Sensor Networks. *Proc. of the IEEE*, 91(8), August 2003.
- [11] P. Levis et. al. TOSSIM: Accurate and Scalable Simulations of Entire TinyOS Applications. In *Sensys*, 2003.
- [12] R. T. Vaughan et. al. On Device Abstractions For Portable, Reusable Robot Code. In *IEEE/RSJ IROS 2003*, Las Vegas, USA, October 2003.
- [13] W. J. Kaiser et al. Networked infomechanical systems (nims) for ambient intelligence. Technical Report CENS TR 0031, Center for Embedded Networked Sensing, UC, Los Angeles, December 2003.
- [14] S. Ganeriwal, R. Kumar, and M. Srivastava. Timing Sync Protocol for Sensor Networks. In *Sensys*, 2003.
- [15] L. Girod, V. Bychkovskiy, J. Elson, and D. Estrin. Locating tiny sensors in time and space: a case study. In *INCCD 2002*, Freiburg, Germany, September 2002.
- [16] J. Heidemann, F. Silva, and D. Estrin. Matching Data Dissemination Algorithms to Application Requirements. In *Sensys*, 2003.
- [17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS 2000*, Cambridge, USA, November 2000.
- [18] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *In MobiCom 2000*, Boston, USA, August 2000.
- [19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Frans Kaashoek. The click modular router. *ACM TOCS*, 18(3):263–297, 2000.
- [20] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the simos machine simulator to study complex computer systems. *ACM TOMACS Special Issue on Computer Simulation*, 1997.
- [21] H. Wang, D. Estrin, and L. Girod. Preprocessing in a Tiered Sensor Network for Habitat Monitoring. In *in EURASIP JASP Special Issue on Sensor Networks*, number 4, pages 392–401, 2003.
- [22] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of IEEE INFOCOM*, 2002.
- [23] Center for embedded networked sensing. www.cens.ucla.edu.
- [24] Crossbow mica2. <http://www.xbow.com>.
- [25] Gnu hurd. <http://www.gnu.org/software/hurd/docs.html>.
- [26] James reserve. <http://cens.ucla.edu/Research/Applications>.
- [27] Ns-2. <http://www.isi.edu/nsnam/ns-documentation.html>.
- [28] Parsec. <http://pcl.cs.ucla.edu/projects/parsec/manual>.
- [29] Qnx. <http://www.qnx.com>.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

SAGE

SAGE is a Special Technical Group (STG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖
- ❖ Aptitude Corporation ❖ Atos Origin B.V. ❖ Delmar Learning ❖
- ❖ DoCoMo Communications Laboratories USA, Inc. ❖
- ❖ Electronic Frontier Foundation ❖ Hewlett-Packard ❖ Interhack Corporation ❖
- ❖ MacConnection ❖ The Measurement Factory ❖ Microsoft Research ❖ Portlock Software ❖
- ❖ Raytheon ❖ Sun Microsystems, Inc. ❖ Taos Mountain, Inc. ❖
- ❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖
- ❖ Microsoft Research ❖ MSB Associates ❖ Raytheon ❖ Ripe NCC ❖ Taos Mountain, Inc. ❖

For more information about membership, conferences, or publications,

see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA

Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-931971-21-8